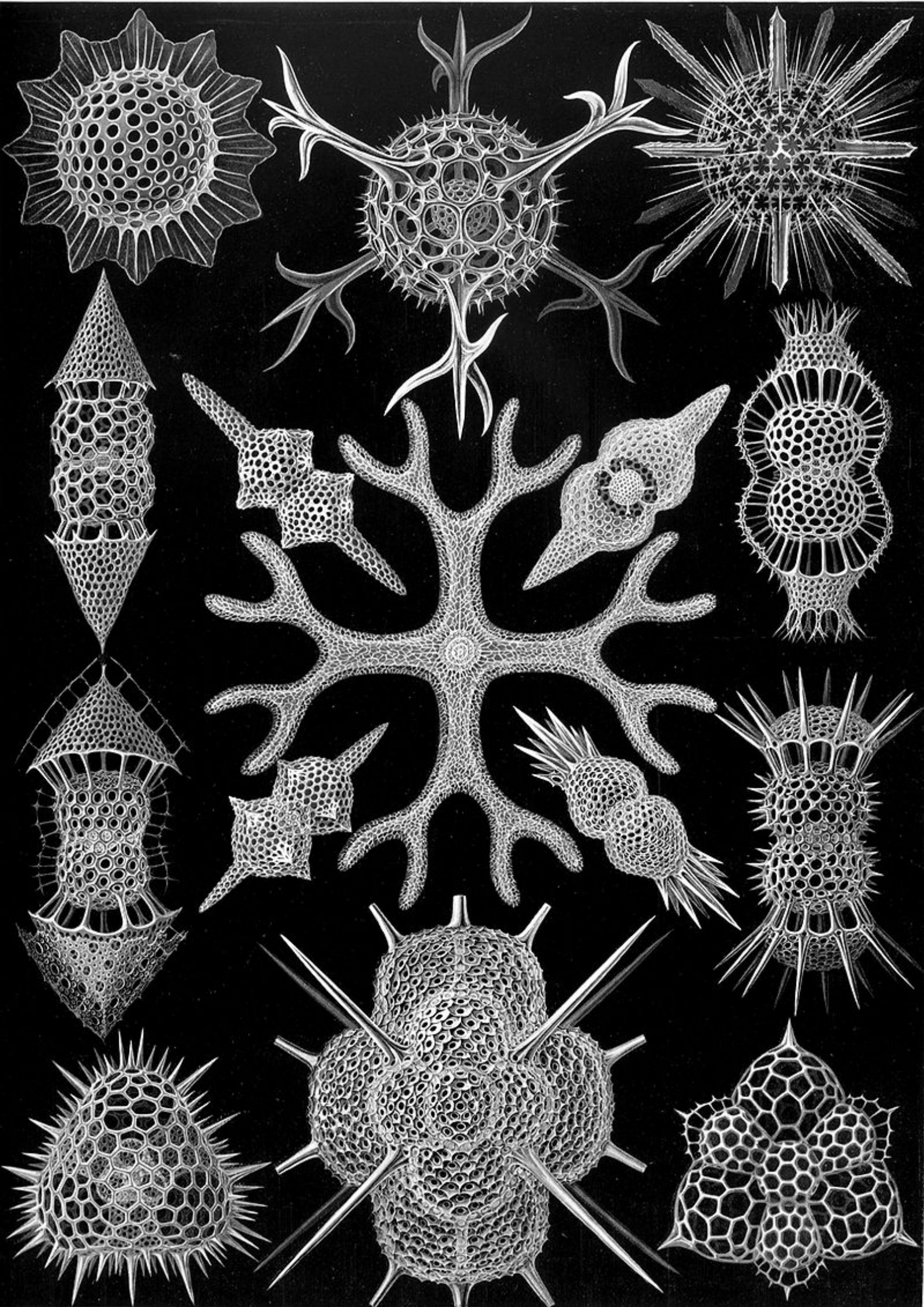
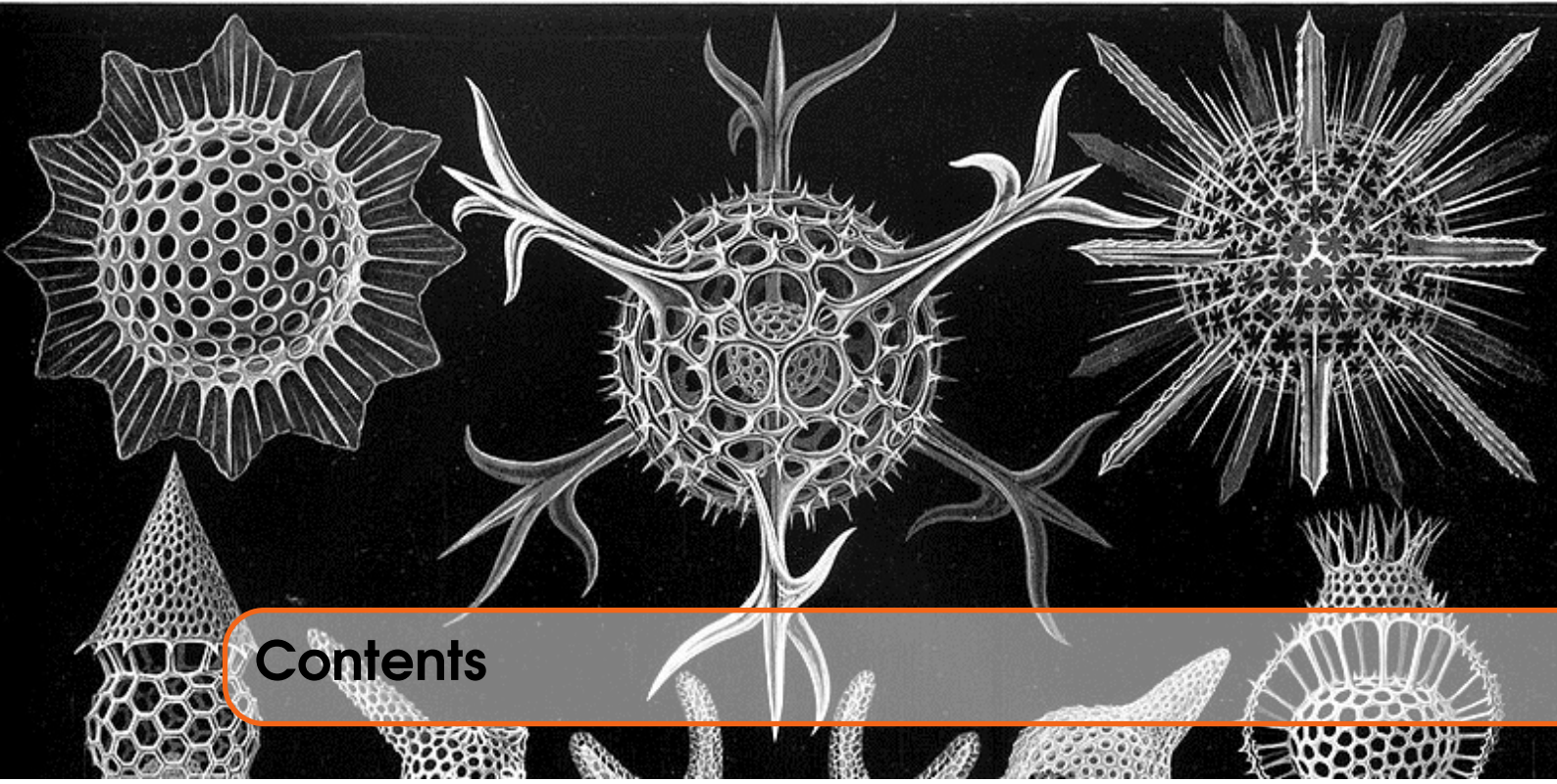


PUBLISHED BY MEMYSELFEYE PUBLISHING ;-)

http://qdosmsq.dunbar-it.co.uk/downloads/AssemblyLanguage/Issue_005/Assembly_Language_005.pdf

This pdf document was created on *16/7/2020* at *09:40:44*.



Contents

1 Preface 9

1.1 Feedback 9

1.2 Subscribing to The Mailing List 9

1.3 Contacting The Mailing List 10

2 Feedback on Issue 4 11

2.1 ASMReformat Comments 11

3 Langton's Ant 13

3.0.1 The Program Listing 13

4 Using the MC68020 - Part 2 27

4.1 Word and Long Memory Access Need Not Be Even! 27

4.2 Bit Field Instructions 28

4.2.1 Bit Fields 28

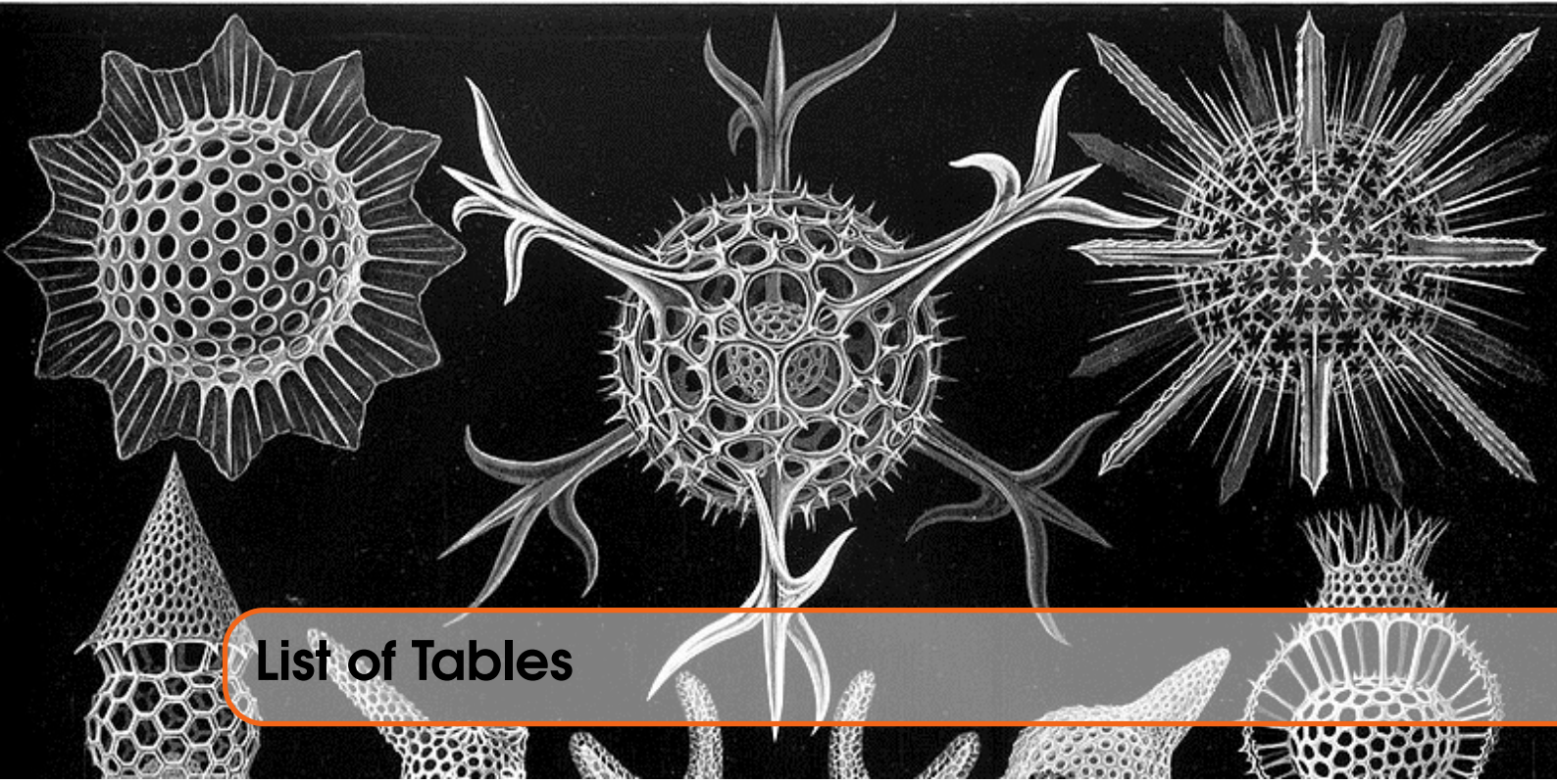
4.2.2 BFCHG - Test Bit Field and Change 30

4.2.3 BFCLR - Test Bit Field and Clear 30

4.2.4 BFEXTS - Extract Bit Field Signed 31

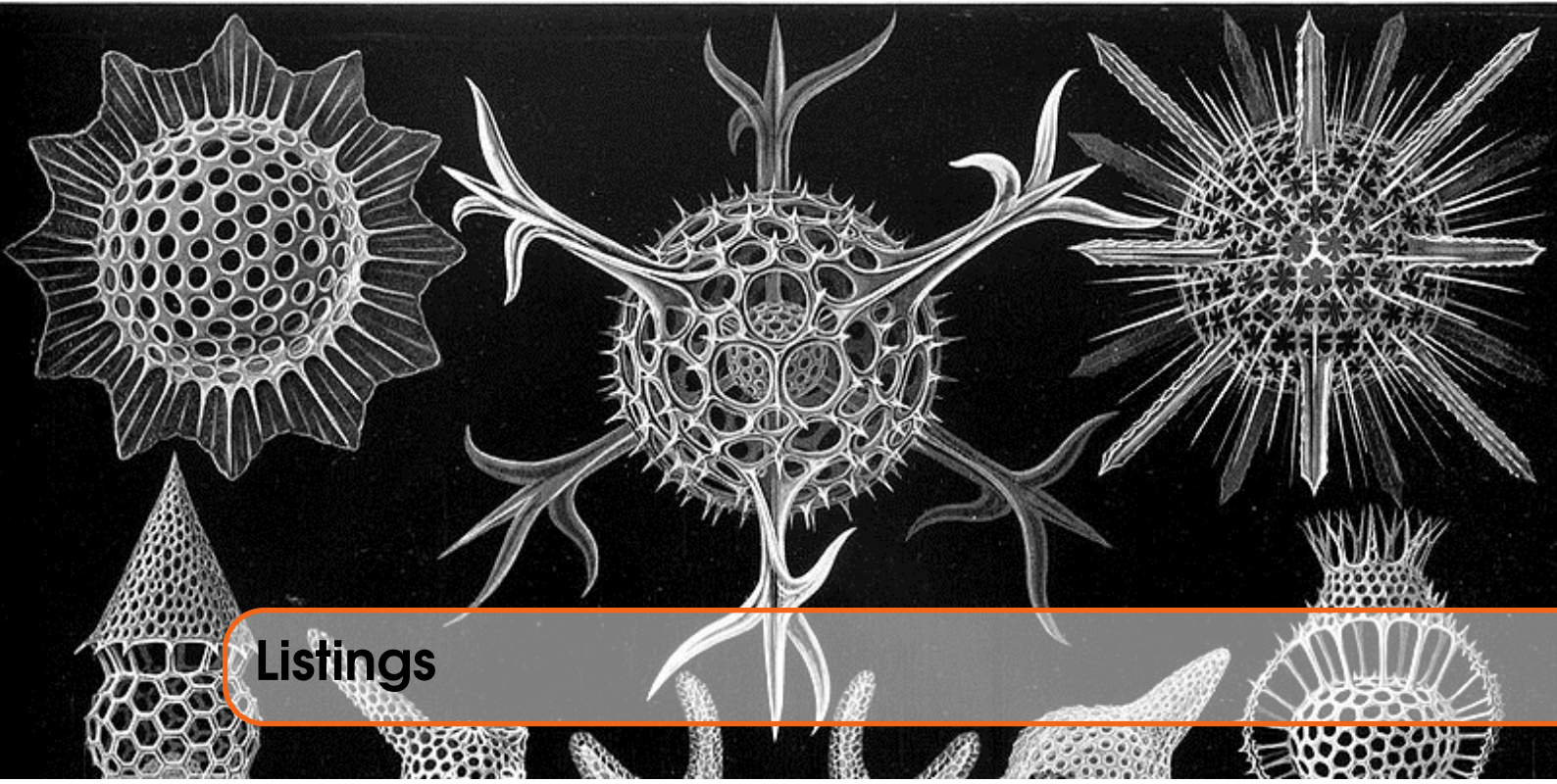
4.2.5 BFEXTU - Extract Bit Field Unsigned 31

4.2.6 BFFFO - Find First One in Bit Field . . . 32
4.2.7 BFINS - Insert Bit Field . . . 32
4.2.8 BFSET - Test Bit Field and Set . . . 33
4.2.9 BFTST - Test Bit Field . . . 33
4.3 Converting Character and Decimal Numbers 33
4.4 Existing Instruction Upgrades 35
4.4.1 Branching and Linking . . . 35
4.4.2 Division . . . 36
4.4.3 Multiplication . . . 36
4.4.4 MOVEC - Move To/From Control Register . . . 37
4.5 Other New Instructions 38
4.5.1 BKPT - Hardware Breakpoint Support . . . 38
4.5.2 CALLM - Call Module . . . 39
4.5.3 CAS and CAS2 - Compare and Swap . . . 39
4.5.4 CHK2 and CMP2 - Check/Compare Register Against Bounds . . . 40
4.5.5 RTM - Return From Module . . . 42
4.5.6 Coprocessor Instructions . . . 42
5 Improving Langton's Ant . . . 43
5.1 A 68020 Improvement, Perhaps? 43
5.1.1 Bit Field Calculations . . . 44
6 This eMagazine is now on Github . . . 47
7 Image Credits . . . 49



List of Tables

4.1 Register Bit Field Conversion Table 29
4.2 5 Byte Bit Field Example 30
4.3 MC68020 Control Registers 37



Listings

3.1 Langtons Ant - Opening Blurb 14
3.2 Langtons Ant - Equates 14
3.3 Langtons Ant - Job Header 15
3.4 Langtons Ant - Command and Channel Definitions 15
3.5 Langtons Ant - Trap Subroutines 16
3.6 Langtons Ant - Start Here 16
3.7 Langtons Ant - Initialise the World 17
3.8 Langtons Ant - Initialising Constants 18
3.9 Langtons Ant - Register Usage 18
3.10 Langtons Ant - Start of Loop 19
3.11 Langtons Ant - Pixel Calculations 20
3.12 Langtons Ant - Rule 1 21
3.13 Langtons Ant - Rule 2 21
3.14 Langtons Ant - New Direction 21
3.15 Langtons Ant - Walk On 22
3.16 Langtons Ant - Plot New Cell Colour 23
3.17 Langtons Ant - Checking ESC 24
3.18 Langtons Ant - Delaying Tactics 24
3.19 Langtons Ant - The Ant's World 24
4.1 How to Blow Up an MC68008 28

4.2 Bit Field Negative Offsets Example 29
4.3 Example of the BFFFO Instruction 32
4.4 Example of the PACK instruction 34
4.5 Example of the UNPK instruction 34
4.6 Example of the CAS Instruction 39
4.7 Example of the CMP2 Instruction 41
5.1 Langtons Ant - Existing Bitmap Calculation 43
5.2 Langtons Ant - 68020 Bitmap Calculation 45
5.3 Langtons Ant - Existing Rule 1 45
5.4 Langtons Ant - Existing Rule 2 45
5.5 Langtons Ant - 68020 Rule 1 46
5.6 Langtons Ant - 68020 Rule 2 46

1. Preface

1.1 Feedback

Please send all feedback to assembly@qdosmsq.dunbar-it.co.uk. You may also send articles to this address, however, please note that anything sent to this email address may be used in a future issue of the eMagazine. Please mark your email clearly if you do not wish this to happen.

This eMagazine is created in LaTeXsource format, aka plain text with a few formatting commands thrown in for good measure, so I can cope with almost any format you might want to send me. As long as I can get plain text out of it, I can convert it to a suitable source format with reasonable ease.

I use a Linux system to generate this eMagazine so I can read most, if not all, Word or MS Office documents, Quill, Plain text, email etc formats. Text87 might be a problem though!

1.2 Subscribing to The Mailing List

This eMagazine is available by subscribing to the mailing list. You do this by sending your favourite browser to http://qdosmsq.dunbar-it.co.uk/mailinglist and clicking on the link "Subscribe to our Newsletters".

On the next screen, you are invited to enter your email address *twice*, and your name. If you wish to receive emails from the mailing list in HTML format then tick the box that offers you that option. Click the Subscribe button.

An email will be sent to you with a link that you must click on to confirm your subscription. Once done, that is all you need to do. The rest is up to me!

1.3 Contacting The Mailing List

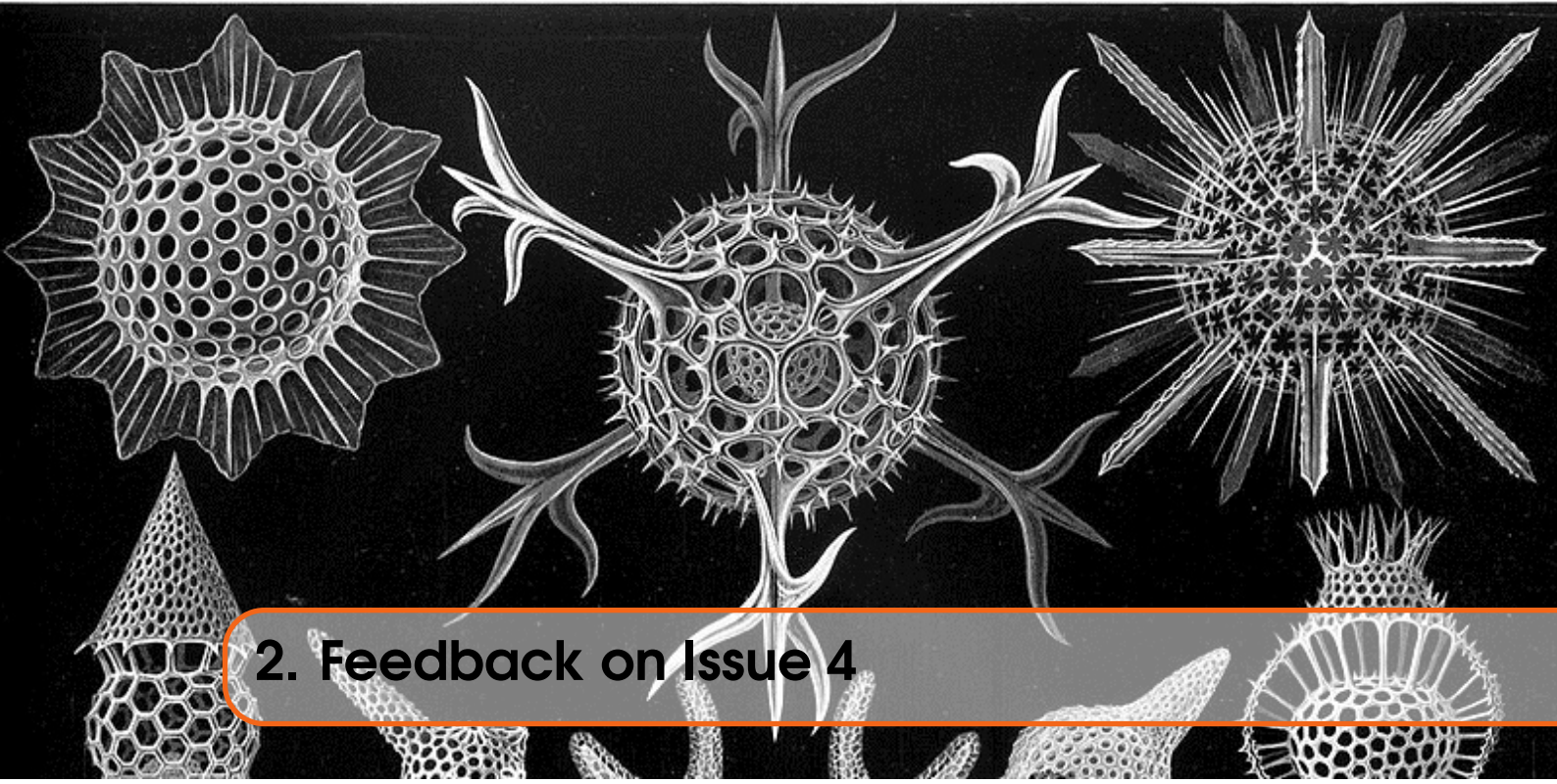
I'm rather hoping that this mailing list will not be a one-way affair, like QL Today appeared to be. I'm very open to suggestions, opinions, articles etc from my readers, otherwise how do I know what I'm doing is right or wrong?

I suspect George will continue to keep me correct on matters where I get stuff completely wrong, as before, and I know George did ask if the list would be contactable, so I've set up an email address for the list, so that you can make comments etc as you wish. The email address is:

assembly@qdosmsq.dunbar-it.co.uk

Any emails sent there will eventually find me. Please note, anything sent to that email address will be considered for publication, so I would appreciate your name at the very least if you intend to send something. If you do not wish your email to be considered for publication, please mark it clearly as such, thanks. I look forward to hearing from you all, from time to time.

If you do have an article to contribute, I'll happily accept it in almost any format - email, text, Word, Libre/Open Office odt, Quill, PC Quill, etc etc. Ideally, a LaTeXsource document is the best format, because I can simply include those directly, but I doubt I'll be getting many of those! But not to worry, if you have something, I'll hopefully manage to include it.



2. Feedback on Issue 4

2.1 ASMReformat Comments

Not long after I published Issue 4 I received some feedback from Wolfgang Lenerz regarding the code for the ASMReformat program listing, well, to be precise, the listing as published *and* the output when run against a source file.

WL: Four traps aren't defined in the code (`io_fline`, `io_sbyte`, `io_sstrg`, `mt_frjob`) you might want to include the `equ` for these.

ND: I did wonder about this when I was writing the utility. GWASS includes a definitions file automatically so when I had those equates in the source, I got errors that they were duplicated, so I had to remove them. I was sure that QMAC did the same thing - but I haven't had to use that assembler since I started writing in QL Today all those years ago.

For anyone who needs them, here are the QDOS versions, from Pennell[1]:

```
io_fline    equ 2
io_sbyte    equ 5
io_sstrg    equ 7
mt_frjob    equ 5
```

WL: I don't know about Gwasl, but Qmac doesn't allow, for the `equ` directive, the label and the content to be on different lines:

```
label equ something
```

is ok

[1] *The Sinclair QDOS Companion* A Guide to the QL Operating System. © Andrew Pennell, 1985

```
label
      equ something
```

is not ok....

ND: Hmm. I wasn't aware of this, but as mentioned above, it's been a long time! Perhaps, in a forthcoming issue, I could amend the utility to accept a parameter string that prevents this line split occurring. I'll look into it and see what I can come up with.

WL: Finally, for Qmac, you need simple "section x" at the start.

ND: This is true, and I *think* you also need an `END` at the end as well, if I remember correctly?

Anyway, I was assuming when I wrote the code that you would be passing your own code through the utility to get proper/my/standard/whatever formatting. so I would assume that you already have a `SECTION x` at the start (and an `END` at the end?) so they should be already there. Unless, of course, you mean that it doesn't work for those lines in a source file?

I've tested it on my own DJToolkit source, which was originally QMAC based, and it works fine.

3. Langton's Ant

This program is something I remember writing many years ago when I was working in Aberdeen,which was some time in the 11 years prior to 1996 when I moved South of the Border, to deprive an Englishman of a woman and a job![1]

Back then, I wrote it in C using *Borland Turbo C++ Version 3.0* - probably the first program I ever wrote in C on a PC.

Anyway, it's a demonstration of something called *Emergent Behaviour* which is something that emerges from some set of rules, but what emerges *was not specifically* programmed into the rules.

The way birds flock together, but manage not to crash into one another, for example, is based on a couple of rules

- Fly in the same direction as your neighbour;
- Don't fly too close nor too far away;
- Don't fly into things.

(Or something similar). A program was written some years ago that embodied those rules and the result was a flock of "boids" as the program was called. It's quite famous.

Anyway, Langton's Ant is another one of those, it has only two rules:

- If you are on a white cell, colour it black, turn right, step forward;
- If you are on a black cell, colour it white, turn left, step forward

That's all there is to it.

3.0.1 The Program Listing

I coded this program in SuperBASIC first of all, just to get my head around the algorithm. When run on QPC on my 10 year old laptop, it ran pretty damned quickly. And yes, the emergent behaviour

[1] Joke!

was there after about 10,000 steps, as expected.[2]

Anyway, I then coded it in Assembly and was wondering just how much faster it would be. The answer? Pretty damned fast. I had to slow it down by inserting a one frame suspension of the job which actually made it run slower than SuperBASIC! But that was then too slow, so I ended up with a delay loop[3] instead.

```
1  ;-------------------------------------------------------------------------
2  ; LANGTON'S ANT
3  ;
4  ; A small routine to demonstrate how emergent behaviour can, ahem,
5  ; emerge from simple rules.
6  ;
7  ; See Wikipedia: https://en.wikipedia.org/wiki/Langton's_ant
8  ;
9  ; The ant lives in a world which is a certain size wide and deep and
10 ; the ends wrap around. It starts off in the middle, and walks in a
11 ; downward direction - it makes no difference though, the behaviour
12 ; will still emerge.
13 ;
14 ; If the ant lands on a cell which is white, it must turn it black,
15 ; turn right, and walk on to the next cell in that new direction.
16 ;
17 ; If the cell was black, the ant must turn it white, turn left, and
18 ; walk on to the next cell in the new direction.
19 ;
20 ; That is all the rules. Run the program and see what happens!
21 ;-------------------------------------------------------------------------
22 ; Norman Dunbar
23 ; 17 February 2018.
24 ;
25 ; This source code is open source. Use it as you see fit and if you
26 ; improve it, tell everyone and let them have the new source!
27 ;-------------------------------------------------------------------------
```

Listing 3.1: Langtons Ant - Opening Blurb

As ever, the code above introduces the program. Not much to see here, but there is a reference to the Wikipedia article about Langton's Ant.

```
28 ;-------------------------------------------------------------------------
29 ; The sizes must be a power of two. We need this later when we limit
30 ; them to anything between 0 and the size minus 1. If they are not a
31 ; power of two, it doesn't work the way I've done things later!
32 ;-------------------------------------------------------------------------
33 size_x
34             equ     512                 ; Pixels across (width)
35 size_y
36             equ     256                 ; Pixels down (height)
37
38 ;-------------------------------------------------------------------------
39 ; Color names, better than guessing the colour values. The ESC key is
40 ; defined here, or its bit number in D1 after an IPC call.
41 ;-------------------------------------------------------------------------
```

[2] See the Wikipedia article at https://en.wikipedia.org/wiki/Langton's_ant

[3] And delay loops burn CPU cycles and keep the computer busy where suspending the job would release resources and let other applications run better.

```
42 black
43              equ     0                       ; Black cell colour
44 red
45              equ     2                       ; It's a red ant!
46 white
47              equ     7                       ; White cell colour
48 green
49              equ     4                       ; The world colour
50 esc
51              equ     3                       ; Bit number for ESC
```

Listing 3.2: Langtons Ant - Equates

The next section of code, above, are the equates that are needed. Please note that the two sizes, `size_x` and `size_y` define the maximum width and height of the world in which the ant lives. They must be a power of two because if they are not, bad stuff will happen later.

Various colours are also defined here, plus the bit that we need to check to see if there was a press of the `ESC` key.

```
52 ;------------------------------------------------------------------------
53 ; Standard Job start.
54 ;------------------------------------------------------------------------
55 start
56              bra.s   Langton
57              dc.l    0
58              dc.w    $4afb
59
60 name
61              dc.w    nameEnd-name-2
62              dc.b    "Langton's Ant"
63 nameEnd
64              equ     *
```

Listing 3.3: Langtons Ant - Job Header

The code above is the standard QDOSMSQ job header. You should be getting used to seeing these by now!

```
66 ;------------------------------------------------------------------------
67 ; BLOCK command parameter block.
68 ;------------------------------------------------------------------------
69 Block
70              dc.w    1,1
71 xPos
72              dc.w    Size_x/2,Size_y/2       ; 256, 128 initial coordinates
73
74 ;------------------------------------------------------------------------
75 ; Check ESC key pressed IPC command string.
76 ;------------------------------------------------------------------------
77 ipc_command
78              dc.b    9,1,0,0,0,0,1,2         ; KEYROW(1)
79
80 ;------------------------------------------------------------------------
81 ; Screen channel name and window definition block.
82 ;------------------------------------------------------------------------
83 scr_def
84              dc.w    4
```

```
85              dc.b    'scr_'                  ; No input needed, so SCR_
86
87  winDef
88              dc.w    size_x
89              dc.w    size_y
90              dc.w    (512-size_x)/2          ; Assumes 512 max width
91              dc.w    (256-size_y)/2          ; Assumes 256 max depth
```

Listing 3.4: Langtons Ant - Command and Channel Definitions

The section of code above sets up a few commands, the first is the `BLOCK` command to draw a 1 pixel by 1 pixel block in the middle of the ant's world.

Next to that is a command to allow the IPC processor to read the keyboard, specifically the equivalent of `KEYROW(1)` which is where we find the `ESC` key.

Finally, we have the definition for a `SCR_` channel and a block of words to redefine the window we will set up on that channel. We do it this way because we only need to set the sizes of the ant's world, and hopefully, the assembler will work out the window definition for us. Lazy? Me?

```
93  ;-------------------------------------------------------------------
94  ; Subroutines to do a trap, test D0 and die horribly if there was
95  ; an error.
96  ;-------------------------------------------------------------------
97  doTrap1
98              trap    #1
99              bra.s   testD0
100
101 doTrap2
102             trap    #2
103             bra.s   testD0
104
105 doTrap3
106             trap    #3
107
108 testD0
109             tst.l   d0
110             bne.s   suicide
111             rts
112
113 suicide
114             move.l  d0,d3
115             moveq   #mt_frjob,d0
116             moveq   #-1,d1
117             trap    #1
```

Listing 3.5: Langtons Ant - Trap Subroutines

The code above is a set of three simple subroutines to execute a `trap` instruction, check the error return and to kill the job if an error occurred.

```
120 ;-------------------------------------------------------------------
121 ; The main starting place. Open a scr_ channel. The channel ID will
122 ; stay protected in A0.L throughout the rest of the code.
123 ;-------------------------------------------------------------------
124 Langton
125             lea     scr_def,a0              ; Screen channel
126             moveq   #io_open,d0
```

```
127             moveq   #-1,d1
128             moveq   #0,d3
129             bsr.s   doTrap2
130
131 ;-------------------------------------------------------------------------
132 ; The screen channel is open, make the window the requested size.
133 ;-------------------------------------------------------------------------
134 Window
135             moveq   #sd_wdef,d0         ; Window definition
136             moveq   #0,d1
137             moveq   #0,d2
138             moveq   #-1,d3
139             lea     winDef,a1
140             bsr.s   doTrap3
141
142 ;-------------------------------------------------------------------------
143 ; Set the paper to green.
144 ;-------------------------------------------------------------------------
145 Paper
146             moveq   #sd_setpa,d0        ; Set paper to green
147             moveq   #green,d1
148             bsr.s   doTrap3
149
150 ;-------------------------------------------------------------------------
151 ; And clear the entire screen.
152 ;-------------------------------------------------------------------------
153 Cls
154             moveq   #sd_clear,d0        ; Cls
155             bsr.s   doTrap3
```

Listing 3.6: Langtons Ant - Start Here

This is the beginning of the real code. We open the channel to the screen, redefine it using the window definition block above, which has been filled in with the appropriate sizes by the assembler, set the paper to green and clear the screen.

Easy stuff, but anyone following http://qlforum.co.uk will know that I had a few problems and needed help. I was calling the wrong `trap` instructions, so instead of opening a channel, I was creating a weird job instead. Not good when you try to set paper colour on a job id rather than a channel id!

```
157 ;-------------------------------------------------------------------------
158 ; The cells bitmap defines the ant's world. A bit is clear for white
159 ; or set for black. Initially, all cells are white. It makes no
160 ; difference - try setting the cells to FF (all black) and you'll see
161 ; the same results.
162 ;-------------------------------------------------------------------------
163 clearCells
164             lea     cells,a3            ; The ant's world lives here
165             move.l  a3,a2               ; Use A2 for the loop
166             moveq   #0,d0               ; Byte counter
167             move.w  winDef(pc),d0       ; Get width in pixels
168             lsr.w   #3,d0               ; Calculate width in bytes
169             mulu    winDef+2(pc),d0     ; Times height in pixels
170             bra.s   clearNext
171
172 clearLoop
```

```
173             clr.b   (a2)+                   ; Clear one byte
174
175 clearNext
176             dbra    d0,clearLoop            ; And the rest
```

Listing 3.7: Langtons Ant - Initialise the World

The ant's world is determined by a bitmap where one bit represents the white or black colour of a cell within the world. The code above clears out the bitmap to ensure that no random bits are to be found. The ant's world is coloured white at the start.

However, feel free to change this to black and fill each byte with $\$FF_{hex}$ instead, if you wish. It is the rules that determine the behaviour.

It should be noted that the size of the ant's world should not exceed a word sized register when multiplying. The calculation is $height * (width/8)$ because each byte represents 8 pixels across.

```
179 ;------------------------------------------------------------------------
180 ; Initialise A2 to point at the block command parameters. We need it
181 ; in A1 actually, but A1 gets corrupted so we save it in A2. We also
182 ; load up the initial position of the ant into D2. The top word of D2
183 ; is the Across coordinate, the low word is the Down coordinate. D5
184 ; is set to zero, for DOWN, which is the direction the ant is facing.
185 ; Try changing it, see what happens.
186 ; A5 is set to the IPC command string, which we need in A3 later. So
187 ; as A3 is preserved through an IPC call, we can EXG these two.
188 ;------------------------------------------------------------------------
189             lea     Block,a2                ; Needed in A1 later
190             move.l  xpos(pc),d2             ; Hi = Across, Lo = Down
191             moveq   #0,d5                   ; Direction ant is walking
192             lea     ipc_command,a5          ; Read KEYROW 1 for ESC key
```

Listing 3.8: Langtons Ant - Initialising Constants

The code above initialises some constants within the program. `A2` which is never corrupted throughout the program, holds the address of the `BLOCK` command. We actually need that in `A1` but that gets corrupted in places, so `A2` is a good place to keep it safe.

`D2` is initialised with the initial coordinates which have been set to be half the width and half the height of the ant's world. The upper word of `D2` is the x or *across* coordinate and the lower word is the y or *down* coordinate.

`D5` holds the direction that the ant is facing. It is initially facing down, but this can be changed if you wish, make sure you set it to a value between zero and three though.

Finally, `A5` holds the address of the IPC command to read the keyboard. This is actually something we need in `A3` but that is needed elsewhere, so `A5` was a handy place to keep it. We will swap those two registers over as and when we need to.

Talking of registers...

```
194 ;------------------------------------------------------------------------
195 ; REGISTERS USED BEYOND THIS POINT
196 ;
197 ; D0 = Trap codes.
198 ; D1 = Ant or Cell colour, red, white or black.
199 ; D2 = Hi word = Across, Lo word = Down coordinates.
200 ; D3 = Timeout for traps. -1 always.
```

```
; D4 = Bit number within Cells bitmap, for the current pixel.
; D5 = Direction indicator. 0=Down, 1=Left, 2=Up, 3=Right.
; D6 = Used to calculate A4 and D4 each time through the loop.
; D7 = Copy of Direction, used in walking to next cell.
;
; A0 = Channel Id
; A1 = Used for trap calls.
; A2 = Pointer to BLOCK command in job code. (Used in A1.)
; A3 = Pointer to Cells bitmap.
; A4 = Pointer to byte in Cells where current coordinates is found.
; A5 = Pointer to IPC command to read KEYROW 1 for ESC key.
; A6 = Reserved for QDOS/SMSQ
; A7 = Stack Pointer.
;
; Cells Bitmap.
;
; The cells array of bytes measures size_y * (size_x / 8) bytes
; and holds 1 bit for every cell in the screen (512 * 256 default)
; with a zero bit meaning a white cell and a 1 bit being black.
;-------------------------------------------------------------------
```

Listing 3.9: Langtons Ant - Register Usage

The comments above show the usage of the registers in the remainder of the program. As you can see, I'm just about using all of them for one thing or another.

```
;-------------------------------------------------------------------
; And here we start the main loop. We adjust the ant coordinates to
; fall into range. This is why we used a power of two as the sizes as
; it makes it easier to keep the coordinates in range 0 to size - 1.
;-------------------------------------------------------------------
antWalk
            andi.w  #size_y-1,d2        ; Make down 0 to size_y-1
            swap    d2
            andi.w  #size_x-1,d2        ; Make across 0 to size_x-1
            swap    d2
            move.l  d2,4(a2)            ; Update block coordinates

;-------------------------------------------------------------------
; We have saved the adjusted coordinates in the BLOCK command's
; parameters, plot the current ant position with a red pixel. Good
; luck seeing that, it moves pretty quickly!
;-------------------------------------------------------------------
plotAnt
            moveq   #sd_fill,d0         ; Block command
            moveq   #red,d1             ; It's a red ant of course!
            move.l  a2,a1               ; Block parameter block
            bsr     doTrap3             ; Won't return on error
```

Listing 3.10: Langtons Ant - Start of Loop

This is the code that runs our ant's world. Because we made the width and height of the world a power of two, we can subtract 1 to get a mask that will ensure that it will remain in range and this also helps in wrapping the ant's world around, top and bottom as well as left and right.

Given that width, for example, is 512, then ANDing the across coordinate with 511 will cause it to be in range 0 to 511 always. If it becomes 512 then and will reset it to 0. If it goes to minus 1, the

and will reset it to 511 - and so, we have a simple way of wrapping the coordinates.

```
245 ;-------------------------------------------------------------------------
246 ; We need to get the current cell at this point. We want the address
247 ; of the current byte in A4 and the number of the bit in that byte in
248 ; D4. At this point we have:
249 ;
250 ; A3 = the cells bitmap start address.
251 ; D2 = Across | Down coordinates.
252 ;-------------------------------------------------------------------------
253
254
255 ;-------------------------------------------------------------------------
256 ; First work out the bit number in the cell's byte. This is simply
257 ; 7 - (across coordinate MOD 8).
258 ;-------------------------------------------------------------------------
259 bitNumber
260             move.l  d2,d6               ; Copy Across | Down coords
261             swap    d6                  ; Down | Across
262             move.w  d6,d4               ; Copy Across coordinate to D4
263             andi.w  #7,d4               ; Mod 8 = 0 to 7
264             neg.w   d4                  ; -7 to 0
265             addq.w  #7,d4               ; D4.W = bit number
266
267 ;-------------------------------------------------------------------------
268 ; The byte within the row, which we calculate soon, is calculated as
269 ; (across coordinate DIV 8). Easy.
270 ;-------------------------------------------------------------------------
271 byteNumber
272             lsr.w   #3,d6               ; Across div 8
273
274 ;-------------------------------------------------------------------------
275 ; The correct row number in the cells bitmap is the Down coordinate
276 ; multipled by (size_x DIV 8). This shouldn't exceed a word size. The
277 ; result is added to A3 which is the start address of the cells
278 ; bitmap and loaded into A4 as the byte address that we want.
279 ;-------------------------------------------------------------------------
280 doRow
281             move.w  #size_x,d7          ; Pixels across
282             lsr.w   #3,d7               ; Now bytes across
283             mulu    d2,d7               ; Times down coordinate
284             lea     0(a3,d7.w),a4       ; Address of correct row
285             adda.w  d6,a4               ; Correct byte in row
286
287 ;-------------------------------------------------------------------------
288 ; Given the byte address in A4 and the bit number in D4, we can now
289 ; test that individual bit. A zero is white while a 1 is black.
290 ;-------------------------------------------------------------------------
291             btst    d4,(a4)             ; 0 = white, 1 = black
```

Listing 3.11: Langtons Ant - Pixel Calculations

The code above takes the two coordinates of the ant's current cell and from them works out which bit in the bitmap represents that cell.

There are four things we need:

- The base address of the cells bitmap. This is held in `A3`.

- The correct "row" in the bitmap; This will be in D7.
- The correct byte within the row; This will be in D6
- The correct bit within the byte; This will be in D4.

The calculations are reasonably simple:

$D4 = 7 - (Across \bmod 8)$

$D6 = (Across \operatorname{div} 8)$

$D7 = Down * (width \operatorname{div} 8)$

Once we have calculated all those, we need simply to load the byte address into A4 and the bit number into D4 and then we can test the bit to see if it is black or white, with zero being white.

```
293 ;-------------------------------------------------------------------------
294 ; Rule 1: If the cell is white, turn it black, right turn, walk on.
295 ;-------------------------------------------------------------------------
296 ruleOne
297             bne.s   ruleTwo
298             bset    d4,(a4)             ; Turn it black in the bitmap
299             moveq   #black,d1           ; Colour for BLOCK command
300             subq.b  #1,d5               ; Direction - 1 = right
301             bra.s   doDirection         ; Prepare to walk on
```

Listing 3.12: Langtons Ant - Rule 1

This is the code for rule 1. If the cell that the ant arrived on is white then colour it black and make a right turn by subtracting 1 from the direction. We don't actually colour the cell here, we simply set the cell's new colour, black, in D1 ready to be plotted below.

```
303 ;-------------------------------------------------------------------------
304 ; Rule 2: If the cell is black, turn it white, left turn, walk on.
305 ;-------------------------------------------------------------------------
306 ruleTwo
307             bclr    d4,(a4)             ; Turn it white in the bitmap
308             moveq   #white,d1           ; Colour for BLOCK command
309             addq.b  #1,d5               ; Direction + 1 = left
```

Listing 3.13: Langtons Ant - Rule 2

This is the code for rule 2. If the cell that the ant arrived on is black then colour it white and make a left turn by adding 1 to the direction. Again there is no actual colouring here, we simply set D1 to be white, ready for later.

```
311 ;-------------------------------------------------------------------------
312 ; We have adjusted the direction. Make sure we restrict it to 0 - 3.
313 ;-------------------------------------------------------------------------
314 doDirection
315             andi.b  #3,d5               ; Restrict to 0 to 3
316             move.b  d5,d7               ; Copy new direction
317
318 ;-------------------------------------------------------------------------
319 ; Ready to walk on in the new direction.
320 ;
321 ; The directions are:
322 ;
323 ;           UP
324 ;           2
```

```
; LEFT 3        1 RIGHT
;           0
;         DOWN
;
; So, if direction is up (2) then turn right = 1 and turn left = 3
;
; Plus 1 = turn left
; Minus 1 = turn right.
;
; If direction = down  = 0 then Down   + 1
; If direction = right = 1 then Across + 1
; If direction = up    = 2 then Down   - 1
; If direction = left  = 3 then Across - 1
;-------------------------------------------------------------------------
```

Listing 3.14: Langtons Ant - New Direction

Most of the code above is simply comments explaining the directions and how adding or subtracting 1 from the current direction equates to a left or right turn. However, we do have to be careful to make sure that the direction, in `D5` is restricted to the values 0 through 3 only.

Because we are going to mess about with the new direction value, in the code below, we copy it to a working register, in this case, `D7`.

```
;-------------------------------------------------------------------------
; Check the direction for Down, if not then skip, otherwise adjust
; the Down coordinate by +1.
;-------------------------------------------------------------------------
doWalk
            bne.s   tryRight

doDown
            addq.w  #1,d2               ; Down + 1
            bra.s   doPlot

;-------------------------------------------------------------------------
; Check if the new direction is Right. If so, adjust the Across
; coordinate by +1.
;-------------------------------------------------------------------------
tryRight
            subq.b  #1,d7
            bne.s   tryUp

doRight
            swap    d2                  ; Down | Across
            addq.w  #1,d2               ; Across + 1
            swap    d2                  ; Across | Down
            bra.s   doPlot

;-------------------------------------------------------------------------
; Check if the new direction is Up. If so, adjust the Down coordinate
; by -1.
;-------------------------------------------------------------------------
tryUp
            subq.b  #1,d7
            bne.s   doLeft              ; Must be left

```

```
373 doUp
374             subq.w  #1,d2                   ; Down - 1
375             bra.s   doPlot
376
377 ;-------------------------------------------------------------------------
378 ; The new direction must be Left. Adjust the Across coordinate by -1.
379 ;-------------------------------------------------------------------------
380 doLeft
381             swap    d2                      ; Down | Across
382             subq.w  #1,d2                   ; Across - 1
383             swap    d2                      ; Across | Down
```

Listing 3.15: Langtons Ant - Walk On

Because we just ANDed the new direction to keep it in range, we might have set the Z flag. If so, the new direction is Down. In that case we add 1 to the current down coordinate in the low word of D2, and skip over the rest of this code section.

Assuming we are not in the downward direction, we check for a heading towards the right. Subtracting 1 will set the Z flag if we are now facing right. If so, we add 1 to the current across coordinate in the high word of D2 and skip the remaining processing.

If we are not facing to the right, are we facing up? Again, subtracting 1 from D7 will set the Z flag. If so, we need to subtract 1 from the ant's current down coordinate in register D2's low word.

Finally, we can only be facing to the left. We have to subtract 1 from the ant's position in the across direction, which is the high word of register D2.

We are now ready to take a step in the new direction as set in D2 but first we have to colour the current cell, which happens to be red at the moment, black or white according to register D1.

```
385 ;-------------------------------------------------------------------------
386 ; We can overwrite the red ant at the current coordinates with either
387 ; a black or a white cell. When we get here, D1 holds the correct
388 ; colour.
389 ;-------------------------------------------------------------------------
390 doPlot
391             moveq   #sd_fill,d0             ; Block command
392             move.l  a2,a1                   ; Block parameter block
393             bsr     doTrap3                 ; Won't return on error
394
395 ;-------------------------------------------------------------------------
396 ; We have overwritten the current coordinate, update the BLOCK
397 ; parameters with the new coordinates - this is effectively a single
398 ; step in the new direction.
399 ;-------------------------------------------------------------------------
400 nowMove
401             move.l  d2,4(a2)                ; Update coordinates
```

Listing 3.16: Langtons Ant - Plot New Cell Colour

The code above plots a white or black cell over the red pixel at the ant's original position[4] with a black or white pixel, depending on which of the two rules were activated by the ant's new position.

Once we have plotted the current cell in the correct black or white colour, we store the new coordinates for the ant in the BLOCK command's parameter, ahem, block. This means that when

[4]To be honest, the program runs so quick that you almost never see the red ant in the original plot but if you slow it right down, you might see it better.

we next pass through the loop, the ant's new coordinates will be plotted in red again, showing the ant's new position.

```
403 ;-------------------------------------------------------------------------
404 ; Read the keyboard and check on the ESC key. If pressed, we are done
405 ; here. If not pressed, go around again after suspending for a few
406 ; frames.
407 ;-------------------------------------------------------------------------
408 hadEnough
409             move.w  d5,-(a7)            ; Direction gets corrupted.
410             exg     a3,a5               ; We need the IPC command in A3
411             moveq   #mt_ipcom,d0        ; IPC command coming up
412             bsr     doTrap1             ; Dies on error
413             move.w  (a7)+,d5            ; Restore direction
414             exg     a3,a5               ; Cells address in A3 again
415             btst    #esc,d1             ; ESC pressed?
416             beq     hangAbout           ; No, pause then go around
417             moveq   #0,d0               ; Yes, show no errors
418             bra     suicide             ; Kill myself
```

Listing 3.17: Langtons Ant - Checking ESC

This section of code checks if the ESC key is being pressed, once around the main loop. Unusually, this call to QDOSSMSQ corrupts register `D5` so we need to preserve it on the stack during the call. It also helps if you restore it afterwards - which I forgot to do and enjoyed many happy hours debugging a strange bug where the code ran for a while, then simply stopped.

```
420 ;-------------------------------------------------------------------------
421 ; Delay processing for a few clocks. Without this, the job has pretty
422 ; much finished by the time you get to see the screen!
423 ;-------------------------------------------------------------------------
424 hangAbout  ;bra     antWalk            ; Uncomment for full speed!
425             moveq   #3,d0               ; Adjust if too slow/fast
426
427 d0Loop      moveq   #-1,d1              ; Inner loop counter
428
429 d1Loop      nop                         ; Delay a bit
430             nop                         ; Delay a bit more
431             dbra    d1,d1Loop           ; End inner loop
432             dbra    d0,d0Loop           ; End outer loop
433
434             bra     antWalk             ; Go around
```

Listing 3.18: Langtons Ant - Delaying Tactics

When I first wrote this assembly version, I didn't have a delay. The code pretty much finished before the first appearance of the screen so I had to slow it down.

Initially I suspended the job for 1 frame, but that slowed it down quite a bit, too slow in fact - my SuperBASIC version was much quicker!

In the end, I slowed it down with a pair of `NOP`s executed $3 * 65536$ times. If your QL or emulator is too slow (or still too fast) adjust the value in `D0` to suit.

If you want to see exactly how fast the code is with no delays, uncomment the code at label `hangAbout` and be amazed!

```
436 ;-------------------------------------------------------------------------
```

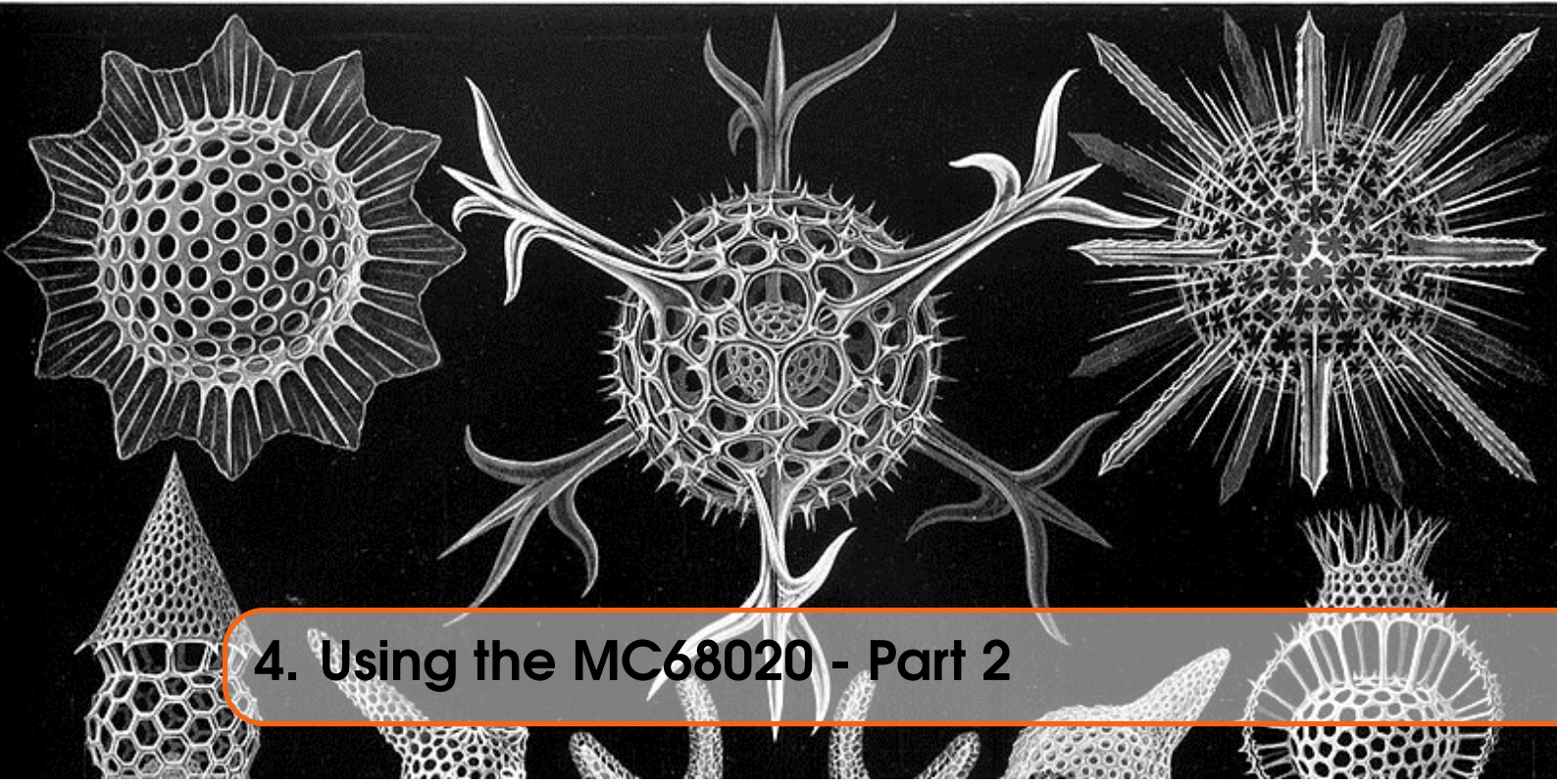
```
437 ; This is the ant's world. It is a bitmap representing the cells that
438 ; the ant walks over. Each bit defines the colour of a cell. Zero is
439 ; white and 1 is black.
440 ;-------------------------------------------------------------------------
441 cells
442             ds.b    size_y*(size_x/8)
```

Listing 3.19: Langtons Ant - The Ant's World

The final part of the listing simply defines a world for the ant. This world consists of one bit per cell and as there are 8 bits (or cells) per byte across the world, we divide by 8 to get the correct number of bytes.

We don't divide for the depth as we need one row per cell in the downward direction.

So that's it. Download the code (or type it in!) from the usual location - http://qdosmsq.dunbar-it.co.uk/downloads/AssemblyLanguage/Issue_005/ - assemble it GWASL works fine - and `EXEC` it. It will run for a bit creating what looks a fairly random pattern on the screen, but then, after about 10,000 iterations of the `antWalk` loop, it will head off in a straight line for no apparent reason. This is the emergent behaviour from two simple rules.



4. Using the MC68020 - Part 2

In the last issue, we took a long look at the addressing modes that are now available when using an MC68020 processor as found in QPC - and possibly, but I don't yet know - in other emulators too. The old BBQL[1] uses an MC68008 and cannot cope with the new stuff.

To assemble these 62020 instructions, you need a copy of Gwass available from George's web site.[2]

One problem I have noticed when using the 68020 instructions, when debugging with QMON, it sometimes gets a tad confused as to exactly which instruction it is executing! Given the age of QMON (and other monitor programs & debuggers) it's hardly surprising that it knows nothing about the 68020, it was current when the 68008 was still *de-rigeur*!

Here are the subjects I will cover in this issue, in relation to the 68020:

- Word and Long memory accessing need no longer be on an even address.
- New bit field instructions.
- Instructions to convert between character and decimal numbers.
- Upgrades to existing instructions.
- New instructions - those not already covered above.

4.1 Word and Long Memory Access Need Not Be Even!

This is something that I've been quietly using and never noticed for a while, at least since QPC was upgraded, with George's help and input, to use a 68020 processor.

Once that happened, any time that I inadvertently accessed a word or long sized memory access, *nothing happened*! In the old days, and on the BBQL itself, the 68008 would have died horribly when that happened.

For example:

[1] Black Box QL
[2] http://gwiltprogs.info/page2.htm

```
1 start
2               lea     oddAddress,a1
3               move.w  (a1),d0
4               ...
5
6 dc.b          0                       ; This should cause an odd address
7
8 oddAddress
9               equ     *
```

Listing 4.1: How to Blow Up an MC68008

Of course, it doesn't really cause the processor a problem, it simply follows its programming and raises an Address Error Exception. Unfortunately, the QL's handling of such an exception is somewhat flawed and this will cause either a complete and utter lock up, or something far worse.

On QPC and on other computers or emulators with an MC68020 processor, the address exception no longer happens.

4.2 Bit Field Instructions

The M68000 family architecture supports variable-length bit field operations on fields of up to 32 bits - in a register - and almost unlimited width in memory. These instructions can be quite handy, as will be shown later, but first, what exactly is a bit field?

4.2.1 Bit Fields

A bit field is simply a number of consecutive bits in an effective address. A bit field is always specified in curly brackets '{' and '}' and there are two parts, the offset and the width, separated by a colon:

`{offset:width}`

The offset is the first bit *numbering starts at zero for bit 31 and ends up at 31 for bit zero - beware!*[3]

You can calculate the desired starting bit number by subtracting the highest bit you want from 31. So, for example, if you want bits 15, 14, 13, 12 and 11 of a register, subtract 15 from 31 to get 16. 16 would be the offset and there are 5 bits, so the bit field specification for the appropriate opcode, would be:

`{16:5}`

For example, in many opcodes, the size of the operand - .B, .W or .L - is specified in bits 5 and 6 of the opcode instruction word in memory.This is therefore a 2 bit wide bit field, starting at bit $31 - 6 = 25$ and would be represented as follows:

`{25:2}`

This would be tagged onto the end of one of the new bit field instructions.

You should note that although the bits number from the left, bit 0 is the *most* significant bit and bit 32 the *least* significant bit, the bit fields start at the given offset and continue *towards the least*

[3] This *really* does my head in! All those years of counting from the right, now I have to start counting from the left *as well*.

significant bit for the defined width. This is the complete reverse of the bit numbering in the registers normally.

Table 4.1 is a handy conversion table. Just look for the bit number in a register that you want to start from and find the bit field offset value in the row above (or below) it.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Table 4.1: Register Bit Field Conversion Table

For example, to start a bit field in register bit 7, we find 7 on the bottom row with 24 above it. So the offset part of our bit field will be 24. Equally, if we wanted to start from register bit 20, we find that on the top row with 11 below it, so 11 is our desired offset. Easy?

Offsets

The offset can be specified as a number from 0 to 31, no other numeric literals are permitted. However, the offset can also be a register name and in that case, all 32 bits can be used, giving an offset of anything from -2^{31} to $2^{32}-1$).

These rather large offsets are *only* for instructions acting on memory locations. It is rather difficult to imagine what a negative offset on a data register, for example, would resolve to!

If an address register, for example, holds a value that is used as the effective address in the instructions, then that is known as the base address. The most significant bit, bit 7, of the base address is offset zero in any bit field. All bit fields are relative to this particular bit.

Negative offsets are fairly simple to use. Assume you have the following code:

```
1  start
2              lea      baseAddress,a1        ; Base address
3              bfextu   (a1){0:8},d1          ; D1 gets $12
4  ;;;;        bfextu   (a1){-32:8},d2        ; WILL NOT ASSEMBLE!
5              move.l   #-32,d2               ; Try again, offset
6              bfextu   (a1){d2:8},d2         ; D2 gets $9A
7              move.l   #24,d7                ; Offset
8              move.l   #4,d6                 ; Width
9              bfextu   (a1){d7:d6},d3        ; D3 gets $07
10             ...
11
12 negAddress
13             dc.l     $9abcdeff          ; Negative offset here
14 baseAddress
15             dc.l     $12345678             ; Base address here
```

Listing 4.2: Bit Field Negative Offsets Example

So the first thing to note is that attempting to use any numeric literals for the offsets will cause assembly errors as they are outside the range 0 to 31 inclusive. This is why we had to use `D2` as the offset and the receiving register for one of the instructions.

The data read into `D1` is the 8 bits from `baseAddress` which is the value $\$12_{hex}$.

The value loaded into register `D2` is from 32 bits *prior* to the address in `A1`, which corresponds to the address of `negAddress`. We therefore get 8 bits from that location loaded into `D2` and that works out at $\$9A_{hex}$.

Finally, D6 and D7 are set up to give a 4 bit width at offset 24 and the 4 bits at that offset from A1 is the nibble $\$7_{hex}$. That value is loaded into D3.

All the values loaded replace the full 32 bit width of the receiving registers.

Widths

Widths are a lot easier than offsets, especially when working with data in RAM. When specified as a number, the width is always between 1 and 32. Any other value results in assembly errors.

When a data register is used to specify the width, the actual width value is taken from the bottom 5 bits of the register and will thus be limited to values between 0 and 31. Hang on! Widths are supposed to be from 1 to 32 bits, not zero to 31. If the register value is zero, then the width is taken to be 32!

When dealing with memory bit fields, the offset and width can cause the bit field to span over 5 separate consecutive bytes in RAM. The processor can cope happily with this. For example:

`bfextu (a1){6:32},d1` spans the 5 bytes from the address in A1 starting at offset 6 of that address (bit 1), passing through the next three bytes, and ending up in the 5th byte at its bit 2 (inclusive).

Byte 1								Byte 2	Byte 3	Byte 4	Byte 5						
7	6	5	4	3	2	1	0	7-0	7-0	7-0	7	6	5	4	3	2	1	0
-	-	-	-	-	-	1	2	3-10	11-18	19-26	27	28	29	30	31	32	-	-

Table 4.2: 5 Byte Bit Field Example

In table 4.2, the top row is the bit numbers in the memory bytes at `(A1)` onwards. The bottom row is the bit numbers in the bit field `{6:32}`. For space reasons the three full bytes in the middle are shortened.

Lets examine the new instructions.

4.2.2 BFCHG - Test Bit Field and Change

The BFCHG instruction sets the condition codes according to the *current* value in a bit field at the specified effective address, then ones-complements the bit field.

For example, a bit field referencing 5 bits starting at bit 5 of D5, would resemble `BFCHG D5{26:5}`.

The instruction tests the bit field first, then sets the flags as follows before changing the bit field:

- X - Not affected.
- N - Set if the most significant bit of the field is set; cleared otherwise.
- Z - Set if all bits of the field are zero; cleared otherwise.
- V - Always cleared.
- C - Always cleared.

4.2.3 BFCLR - Test Bit Field and Clear

The BFCLR instruction Sets condition codes according to the value in a bit field at the specified effective address and finally clears the specified bit field.

The instruction tests the bit field and sets the flags as follows:

- X - Not affected.

- N - Set if the most significant bit of the field is set; cleared otherwise.
- Z - Set if all bits of the field are zero; cleared otherwise.
- V - Always cleared.
- C - Always cleared.

An example to clear the three bits 6, 5 and 4 of the memory address that `A1` points to, would be:

```
BFCLR (A1){1:3}
```

4.2.4 `BFEXTS` - Extract Bit Field Signed

The `BFEXTS` instruction extracts a bit field from the specified effective address location, sign extends it to 32 bits, and loads the result into the destination data register.

The instruction sets the flags as follows:

- X - Not affected.
- N - Set if the most significant bit of the field is set; cleared otherwise.
- Z - Set if all bits of the field are zero; cleared otherwise.
- V - Always cleared.
- C - Always cleared.

For example, to extract the three bits 6, 5 and 4 of the memory address that `A1` points to, into register `D0`, with bit 6 indicating the sign, would be:

```
BFEXTS (A1){1:3},D0
```

The flags are set according to the value in the bit field moved into `D0`, in this case. Bit 4 of the byte would end up in bit zero of `D0`, bit 5 in bit 1 and bit 6 would be sign extended into all bits from bit 2 to bit 31 (in the normal manner of numbering bits).

4.2.5 `BFEXTU` - Extract Bit Field Unsigned

The `BFEXTU` instruction is similar to the `BFEXTS` instruction above, but is unsigned. It extracts a bit field from the specified effective address location, zero extends it to 32 bits, and loads the results into the destination data register.

The instruction sets the flags as follows:

- X - Not affected.
- N - Set if the most significant bit of the field is set; cleared otherwise.
- Z - Set if all bits of the field are zero; cleared otherwise.
- V - Always cleared.
- C - Always cleared.

For example, to extract the three bits 6, 5 and 4 of the memory address that A1 points to, into register `D0`, would be:

```
BFEXTU (A1){1:3},D0
```

The flags are set according to the value in the bit field moved into `D0`, in this case. Bit 4 of the byte would end up in bit zero of `D0`, bit 5 in bit 1 etc.

4.2.6 BFFFO - Find First One in Bit Field

The BFFFO instruction searches the source operand for the most significant bit that is set to a value of one.

The bit offset of that bit (the bit field *offset* in the instruction plus the *offset* of the first one bit in the bit field) is placed in the specified data register.

If no bits in the bit field are set, the result is the field offset plus the field width.

The instruction sets the flags as follows:

- X - Not affected.
- N - Set if the most significant bit of the field is set; cleared otherwise.
- Z - Set if all bits of the field are zero; cleared otherwise.
- V - Always cleared.
- C - Always cleared.

An example to find the first set bit within the 9 bits of register D0, starting at bit 15 and extending down to bit 6, setting the result in register D1, would be:

`BFFFO D0{16:9},D1.`

If there were no set bits in the bit field, the result in D0 would be 16+9 or 25, otherwise it would be 16+ < bit number of set bit >. Bear in mind, the first one bit is the bit number *in the bit field* and not the bit number in the register or memory location. The bit numbers in the bit field start from zero and increase up to the field width minus 1.

This example should help:

```
1  start
2              clr.l    d0                  ; No bits set in D0
3              bfffo    d0{16:9},d1         ; No bits set, so D1=16+9=25
4
5  ;                                          {16:9} ---- ---- - (Bit Field)
6  ;                                                 0123 4567 8 (BF Offset)
7              move.l   #$300,d0            ; $0000  0000 0011 0000 0000
8              bfffo    d0,{16:9},d1        ; Bit 6 of BF set so D1=16+6=22
```

Listing 4.3: Example of the BFFFO Instruction

The result is the bit field start plus the offset into the bit field where the most significant set bit is found. In our bit field, the starting offset is 16. The first set bit in the 9 bits making up the width of the bit field, is bit number 6 - starting at zero - *in the bit field*, so the result is 16+6 or 22.

A valid range of values in D1 would be 16+0 through 16+8 if a single bit in any position of the bit field was set, or 16+9 if no bits in the bit field were set.

If you need to find out exactly which bit in the register that relates to, subtract the result from 31. In this example, 31 − 22 gives 9 and indeed, it is bit 9 in the register which is the most significant set bit.

4.2.7 BFINS - Insert Bit Field

The BFINS instruction inserts a bit field taken from the low-order bits of the specified data register into a bit field at the effective address location. This means that if you have a 3 bit wide bit field in the effective address, then the lowest three bits of the specified data register will be copied to that bit field.

The instruction sets the flags as follows:

- X - Not affected.
- N - Set if the most significant bit of the field is set; cleared otherwise.
- Z - Set if all bits of the field are zero; cleared otherwise.
- V - Always cleared.
- C - Always cleared.

Continuing the example from above, to move the three bits of D0 into a three bit wide bit field of register D4 in bits 6, 5 and 4, we would use the instruction:

```
BFINS D0,D4{25:3}
```

Bit 0 of D0 would end up in bit 4 of D4, bit 1 would be in bit 5 and bit 2 would end up in bit 6.

4.2.8 BFSET - Test Bit Field and Set

The BFSET instruction tests the bit field and sets the flags as follows and then sets all the bits in the field to 1.

- X - Not affected.
- N - Set if the most significant bit of the field is set; cleared otherwise.
- Z - Set if all bits of the field are zero; cleared otherwise.
- V - Always cleared.
- C - Always cleared.

For example, to set all the bits in bits 6 to 9 (a total of 4 bits) of register D0, we would use the instruction:

```
BFSET D0{25:4}
```

4.2.9 BFTST - Test Bit Field

The BFTST instruction tests the specified bit field and sets the flags as follows:

- X - Not affected.
- N - Set if the most significant bit of the field is set; cleared otherwise.
- Z - Set if all bits of the field are zero; cleared otherwise.
- V - Always cleared.
- C - Always cleared.

For example, to test the bits in the 3 bit wide bit field beginning at bit 13 of D4 (that's bit 13 numbering in the normal manner) we need to specify the following instruction:

```
BFTST D4{18:3}
```

4.3 Converting Character and Decimal Numbers

The BBQL and the MC68008 has always had the ability to add, subtract and negate Binary Coded Decimal numbers. However, it was never easy to create such things as there were no instructions that would convert an ASCII (or EBCDIC) string of digits, for example, into a BCD format, and vice versa.

With the MC68020 we have the PACK and UNPK instructions to help us do exactly that.

For other character code sets, the instructions take an immediate data portion, which is added to the characters when encoding, and subtracted when decoding. For ASCII or EBCDIC, this immediate data must be zero.

The PACK instruction has the two following formats:

```
PACK Ds,Dd,#data
PACK -(As),-(Ad),#data
```

The Ds or (As) registers point at the source text, while the Dd or (Ad) registers point at the destination.

The instruction will only convert 2 bytes of the source into a single byte in the destination.

For example, if using the data register version, (D0.L) contains 31323334_{hex}, which is the ASCII character representation of 1234, it takes up the entire 32 bits of the register. All those '3' nibbles are effectively wasted space.

The instruction PACK D0,D1,#0 will convert the ASCII in D0 into the BCD value 34_{hex} in D1.B.

To convert a string of digits, you need to use the in memory version, or do a few shifts along the way.

```
1  start
2              lea      asciiDigits+8,a1      ; Source ASCII string
3              lea      bcdDigits+4,a2        ; Destination BCD buffer
4              moveq    #3,d2                 ; 4 pairs of digits to convert
5
6  loop
7              pack     -(a1),-(a2),#0        ; Convert two bytes
8              dbra     d2,loop               ; Loop for more
9              ...                            ; Done, bcdDigits = $12345678
10
11 asciiDigits
12             dc.b     '12345678'            ; = $3132333435363738
13
14 bcdDigits
15             ds.l     1                     ; Space for result
```

Listing 4.4: Example of the PACK instruction

This code converts the 8 ASCII digits at asciiDigits - in reverse order - from 3132333435363738_{hex} into a long word at bcdDigits which will end up containing the value 12345678_{hex} thus, packing 64 bits of ASCII data into 32 bits of BCD data.

The UNPK instruction has two formats:

```
UNPK Ds,Dd,#data
UNPK -(As),-(Ad),#data
```

And is simply the reverse of the PACK instruction. It converts a single byte into two separate ASCII (or EBCDIC) characters.

```
1  start
2              lea      bcdDigits+4,a1        ; Source BCD buffer
3              lea      asciiDigits+8,a2      ; Destination ASCII string
4              moveq    #3,d2                 ; 4 BCD digits to convert
5
6  loop
7              unpk     -(a1),-(a2),#0        ; Convert two bytes
```

```
8               dbra    d2,loop             ; Loop for more
9               ...                         ; Done, asciiDigits = 12345678
10
11  asciiDigits
12              ds.b    8                   ; Space for ASCII result
13
14  bcdDigits
15              dc.l    $12345678           ; Source BCD number
```

Listing 4.5: Example of the UNPK instruction

Once again, the data are converted in reverse so the initial buffer pointers have to be one past the end of the buffers' last character.

The long word at `bcdDigits`, 12345678_{hex} is converted back into the string 3132333435363738_{hex} or ASCII '12345678' by the above code.

4.4 Existing Instruction Upgrades

Some instructions have changed since their usage in the good old MC68008 installed in the original BBQL. These are discussed below.

4.4.1 Branching and Linking

Currently, on the BBQL, we can have short (8 bit) or word branches in the `Bcc`, `BRA` and `BSR` instructions. These are two's compliment (aka signed) displacements allowing for forward and backward branching. In the MC68020 we now have a long sized branch as well with all 32 bits being permitted.

Using `BSR` as an example, we now have the following:

`BSR.S` and `BSR.B` - this form always assembles to an 8 bit displacement branch which requires two bytes in the binary code.

`BSR.W` - this form always assembles to a 16 bit displacement branch which requires four bytes in the binary code.

`BSR.L` - this form always assembles to a 32 bit displacement branch which requires six bytes in the binary code.

If you don't specify a size for the branch instruction, then it depends on whether your assembler has been configured to always use a 16 bit displacement or if you, like me, have configured it to try an 8 bit displacement and error out if the displacement is too far.

Note, the various *Decrement and Branch Unless Condition* (`DBcc`) instructions have *not* changed, they are restricted to a word sized, 16 bit displacement as before.

The `LINK` instruction now also has a 32 bit displacement. If you use the old instruction `LINK An,#displacement`, you continue to get the 16 bit displacement version. You can now. however, also specify a size of .W to force a 16 bit displacement. `LINK.W An,#displacement`.

To force a 32 bit displacement, specify a size of .L as in `LINK.L An,#displacement`.

4.4.2 Division

On the BBQL we have played around with the `DIVU` and `DIVS` instructions which are word sized in that they return a pair of 16 bit values one for the quotient and one for the remainder. Effectively, a 32 bit value is divided by a 16 bit value to return a pair of 16 bit values in a single data register. The top word is the remainder and the low word is the quotient.

The format of these instructions was always like `DIVS #1234,D0` and there was no need to specify a size - .B, .W or .L - because it was always .W.

On the MC68020 we now have the ability to divide 32 and 64 bit numbers by 32 bit ones, resulting in a 32 bit number for the quotient and another for the remainder. We still have the existing word sized divides, but now you need to specify a size of '.W' to indicate your wish to use the old style. Assemblers will still accept the old style of missing out the size specifier and act accordingly to assemble the old word sized instructions.

There are a further three assembly formats for the divide instructions:

`DIVS <EA>,Dn` - This form divides a 32 bit long word by another 32 bit long word. The result is a 32 bit quotient in the specified register, with the remainder being discarded. A handy integer division instruction.

`DIVS <EA>,Dr:Dq` - This form divides a 64 bit quad word (in any two data registers) by a 32 bit word taken from the effective addresss. The result is a long-word quotient in the `Dq` register, and a long-word remainder in the `Dr` register.

`DIVSL.L <EA>,Dr:Dq` - This form divides a 32 bit long word by another 32 bit long word. The result is a 32 bit quotient in the `Dq` register, and a 32 bit remainder in the `Dr` register. You should note that this instructions has an 'L' tagged on as well as the '.L' for the size. `DIVSL`.

Although I've used `DIVS` in the above examples, the `DIVU` instructions are identical.

Two special conditions may arise during division:

- Division by zero causes a trap.
- Overflow may be detected and set *before* the instruction completes. If the instruction detects an overflow, it sets the overflow condition code, and the operands are unaffected.

The flags are set as follows:

- X - Not affected.
- N - Set if the quotient is negative; cleared otherwise; undefined if overflow or divide by zero occurs.
- Z - Set if the quotient is zero; cleared otherwise; undefined if overflow or divide by zero occurs.
- V - Set if division overflow occurs; undefined if divide by zero occurs; cleared otherwise.
- C - Always cleared.

4.4.3 Multiplication

Upgrades to the `MULU` and `MULS` instructions are similar to those of the division instructions above. Whereas up until now we have been limited to multiplying two 16 bit words to get a 32 bit result, we now have the ability to multiply two 32 bit long words together to give either a 32 bit long word result, or a 64 bit quad word result.

`MULS.L <EA>,Dn` - this form multiplies the 32 bit effective address value by the data register

and stores the lowest 32 bits of the result in the data register. The upper 32 bits of the result are discarded.

`MULS.L <EA>,Dh-Dl` - this form multiplies the 32 bit effective address value by the data register and stores the highest 32 bits of theb result in the `Dh` register and the lowest 32 bits of the result in the `Dl` register. Nothing is discared.

The multiplication instructions set the flags as follows:

- X - Not affected.
- N - Set if the result is negative; cleared otherwise.
- Z - Set if the result is zero; cleared otherwise.
- V - Set if overflow; cleared otherwise.
- C - Always cleared.

For `MULS`, overflow, setting V = 1, can occur only when multiplying 32-bit operands to yield a 32-bit result. Overflow occurs if the highest 32 bits of the quad-word product are not the sign extension of the lowest 32 bits.

For `MULU`, overflow, setting V = 1, can occur only when multiplying 32-bit operands to yield a 32-bit result. Overflow occurs if any of the high-order 32 bits of the quad-word product are not equal to zero.

Overflow cannot occur when the product is a 64 bit quad word.

4.4.4 `MOVEC` - Move To/From Control Register

This instruction is only available on the has two separate formats:

`MOVEC Rc,Rn` `MOVEC Rn,Rc`

The instruction is privileged, so must be run in supervisor mode.

Register `Rc` is a *control* register, see table 4.3, while register `Rn` is a normal register.

Moves the contents of the specified control register (Rc) to the specified general register (Rn) or copies the contents of the specified general register to the specified control register.

This is always a 32-bit transfer, even though the control register may be implemented with fewer bits. Unimplemented bits are read as zeros.

What is a control register? On the MC68020 we have the following control registers:

Control Register	Description
SFC	Source Function Code
DFC	Destination Function Code
USP	User Stack Pointer
VBR	Vector Base Register
CACR	Cache Control Register
CAAR	Cache Address Register
MSP	Master Stack Pointer
ISP	Interrupt Stack Pointer

Table 4.3: MC68020 Control Registers

The flags are not affected by these instructions.

- X - Not affected.
- N - Not affected.
- Z - Not affected.
- V - Not affected.
- C - Not affected.

4.5 Other New Instructions

4.5.1 `BKPT` - Hardware Breakpoint Support

This is a new instruction (from the MC68010 onwards) which is useful for hardware debuggers and is unlikely to be of any use in QL programs. However, I have been wrong in the past!

The manual has this to say about the instruction:

For the MC68010, a breakpoint acknowledge bus cycle is run with function codes driven high and zeros on all address lines. Whether the breakpoint acknowledge bus cycle is terminated with $\overline{DTACK}$, $\overline{BERR}$, or $\overline{VPA}$, the processor always takes an illegal instruction exception. During exception processing, a debug monitor can distinguish different software breakpoints by decoding the field in the `BKPT` instruction.

For the MC68000 and MC68008[4]*, the breakpoint cycle is not run, but an illegal instruction exception is taken.*

For the MC68020, MC68030, and CPU32, a breakpoint acknowledge bus cycle is executed with the immediate data (value 0 – 7) on bits 2 – 4 of the address bus and zeros on bits 0 and 1 of the address bus. The breakpoint acknowledge bus cycle accesses the CPU space, addressing type 0, and provides the breakpoint number specified by the instruction on address lines A2 – A4. If the external hardware terminates the cycle with $\overline{DSACKx}$ or $\overline{STERM}$, the data on the bus (an instruction word) is inserted into the instruction pipe and is executed after the breakpoint instruction. The breakpoint instruction requires a word to be transferred so, if the first bus cycle accesses an 8 bit port, a second bus cycle is required. If the external logic terminates the breakpoint acknowledge bus cycle with $\overline{BERR}$ (i.e. no instruction word available), the processor takes an illegal instruction exception.

For the MC68040, this instruction executes a breakpoint acknowledge bus cycle. Regardless of the cycle termination, the MC68040 takes an illegal instruction exception.

For more information on the breakpoint instruction refer to the appropriate user's manual on bus operation.

This instruction supports breakpoints for debug monitors and real-time hardware emulators.

So that's clear then? It's hardware, only Dave and Nasta (and a few others) will understand any of the above!

The flags are not affected.

- X - Not affected.
- N - Not affected.
- Z - Not affected.
- V - Not affected.
- C - Not affected.

[4] Excuse me? My 68008 manual, a Motorola official one, makes no mention of this instruction for either the MC68008 or the MC68010!

4.5.2 CALLM - Call Module

The CALLM instruction saves the *Current Module State* on the stack and loads a new module state from the destination.

The format of the instruction is:

```
CALLM #data,<EA>
```

This instruction, and RTM require external hardware to be effective.

The effective address of the instruction is the location of an external module descriptor.

A module frame is created on the top of the stack, and the current module state is saved in the frame. (Program counter etc.)

The immediate operand specifies the number of bytes of arguments to be passed to the called module and allows RTM to tidy the stack afterwards.

A new module state is loaded from the descriptor addressed by the effective address.

The flags are not affected.

- X - Not affected.
- N - Not affected.
- Z - Not affected.
- V - Not affected.
- C - Not affected.

See also RTM which is used to return from a CALLM. Also note that these two instructions have been removed from the MC68040 onwards.

4.5.3 CAS and CAS2 - Compare and Swap

The format of the CAS instruction is:

```
CAS.size Dc,Du,<EA>
```

The size can be .B, .W or .L.

The CAS, Compare and Swap, instruction subtracts the value in the 'compare' register (Dc) from the effective address which is the destination also, and sets the condition codes accordingly.

If the Z flag gets set, the value in the 'update' register (Du) is moved to the destination.

If the Z flag was not set by the instruction, the contents of the effective address are moved to the Dc register.

The CAS instruction sets the flags as follows:

- X - Not affected.
- N - Set if the most significant bit of the field is set; cleared otherwise.
- Z - Set if the result is zero; cleared otherwise.
- V - Set if the compare generates an overflow; cleared otherwise.
- C - Set if the compare generates a carry; cleared otherwise.

An example of the CAS instruction is as follows. I admit it's a little contrived, but it shows what happens.

```
1 start
2         lea     label,a1          ; Where the compared data lives
```

```
3               moveq     #2,d2                 ; Dc = compare register
4               move.l    #$87654321,d3         ; Du = update register
5               cas.l     d2,d3,(a1)            ; Compare (A1) with D2
6   ;                                           ; IF (A1) = D2 THEN
7   ;                                           ;    LET (A1) = D3
8   ;                                           ; ELSE
9   ;                                           ;    LET D2 = (A1)
10  ;                                           ; END IF
11              cas.l     d2,d3,(a1)            ; Compare (A1) with D2 again
12              ...
13  label
14          dc.l      $1234
```

Listing 4.6: Example of the CAS Instruction

So, in the above, the first CAS compares the value at (A1) which is 00001234_{hex}$ with the value in D2 which is 00000002_{hex}$ and they are not equal, so D2 gets loaded with the value at (A1) and becomes 00001234_{hex}$.

The second CAS compares the value at (A1) which is still 00001234_{hex}$ with the value in D2 which is now also 00001234_{hex}$ and they are obviously equal, so the long word at (A1) gets set to the value in D3 which happens to be 87654321_{hex}$.

The CAS2 instruction is similar, except that it uses two 'compare' registers and two 'update' registers. The format is:

```
CAS2.size Dc1:Dc2,Du1:Du2,(Rn1):(Rn2)
```

This instruction compares memory operand 1 (Rn1) to compare operand 1 (Dc1). If the operands are equal, the instruction compares memory operand 2 (Rn2) to compare operand 2 (Dc2). If these operands are also equal, the instruction writes the update operands (Du1 and Du2) to the memory operands (Rn1 and Rn2).

If either comparison fails, the instruction writes the memory operands (Rn1 and Rn2) to the compare operands (Dc1 and Dc2).

The size can be .W or .L.

The CAS2 instruction sets the flags as follows:

- X - Not affected.
- N - Set if the most significant bit of the field is set; cleared otherwise.
- Z - Set if the result is zero; cleared otherwise.
- V - Set if the compare generates an overflow; cleared otherwise.
- C - Set if the compare generates a carry; cleared otherwise.

Both CAS and CAS2 instructions access memory using locked or read-modify-write transfer sequences, providing a means of synchronizing several processors in a multi-processor system.

4.5.4 CHK2 and CMP2 - Check/Compare Register Against Bounds

The CHK2 and CMP2 are similar instructions except that while the latter simply sets the condition codes in the status register depending on the results of the comparison, the former traps out through an exception vector. A tad excessive if you ask me! ;-)

The CHK2 instruction has the format:

```
CHK2.size <EA>,Rn
```

It looks in memory at the effective address specified for two bytes, words or long words and compares the specified register, which can be a data or address register, with the two values.

The two values are the (signed) lower and upper bounds for the comparison. The data at the address specified in the effective address is the lower bound. The data at that address plus 1, 2 or 4, depending on the size, is the upper bound.

The flags are set as follows:

- X - Not affected.
- N - Undefined.
- Z - Set if Rn equals either boundary value; cleared otherwise.
- V - Undefined.
- C - Set is Rn is out of range; cleared otherwise.

In operation, if the register value is equal to, or falls between the two bounds, then execution continues as normal and the C flag is cleared. If the register value happens to equal one of the bounds, then the Z flag will be set.

If the register value is out of bounds, then the exception is raised. A tad harsh, so let's look at a gentler approach. The `CMP2` instruction.

This instruction takes the format:

`CMP2.size <EA>,Rn`

It sets the flags exactly as the `CHK2` instruction does.

It operates exactly as the `CHK2` instruction just described, but does not raise an exception if the value is out of bounds. This can be useful when comparing a value in a register to determine if it is a digit, for example, as per the following snippet:

```
1  start
2              lea      inputBuffer,a1        ; Where we find user input
3
4  checkLoop
5              move.b   (a1)+,d0              ; Grab one character
6              cmp2.b   digitBounds,d0        ; Got a digit?
7              bcc.s    gotDigit              ; Yes, handle it
8              cmpi.b   #10,d0                ; No, check for a linefeed?
9              beq.s    allDone               ; End of input
10             ...                            ; Not a digit, do something else
11             bra.s    checkLoop             ; Keep going
12
13 allDone
14             moveq    #0,d0                 ; Signal done
15             rts                            ; Back to caller
16
17 gotDigits
18             ...                            ; Handle digit input here
19             bra.s    checkLoop             ; Loop around
20
21 digitBounds
22             dc.b     '0','9'               ; Range of valid digits
```

Listing 4.7: Example of the `CMP2` Instruction

This is much less hassle than checking the character in `D0` with a '0' and skipping out if the flags show that `D0` was lower than a '0' then doing something similar for a '9'. I also happen to think

that it's a lot easier to read the code done in this way.

4.5.5 `RTM` - Return From Module

This instruction, and `CALLM` require external hardware to be effective.

This instruction, is used to terminate a `CALLM` instruction. It reloads the module state from the stack. The instruction format is:

```
RTM Rn
```

The register can be a data or address register.

A previously saved module state is reloaded from the top of stack. After the module state consisting of program counter, status word etc, is retrieved from the top of the stack, the caller's stack pointer is incremented by the argument count value in the module state.

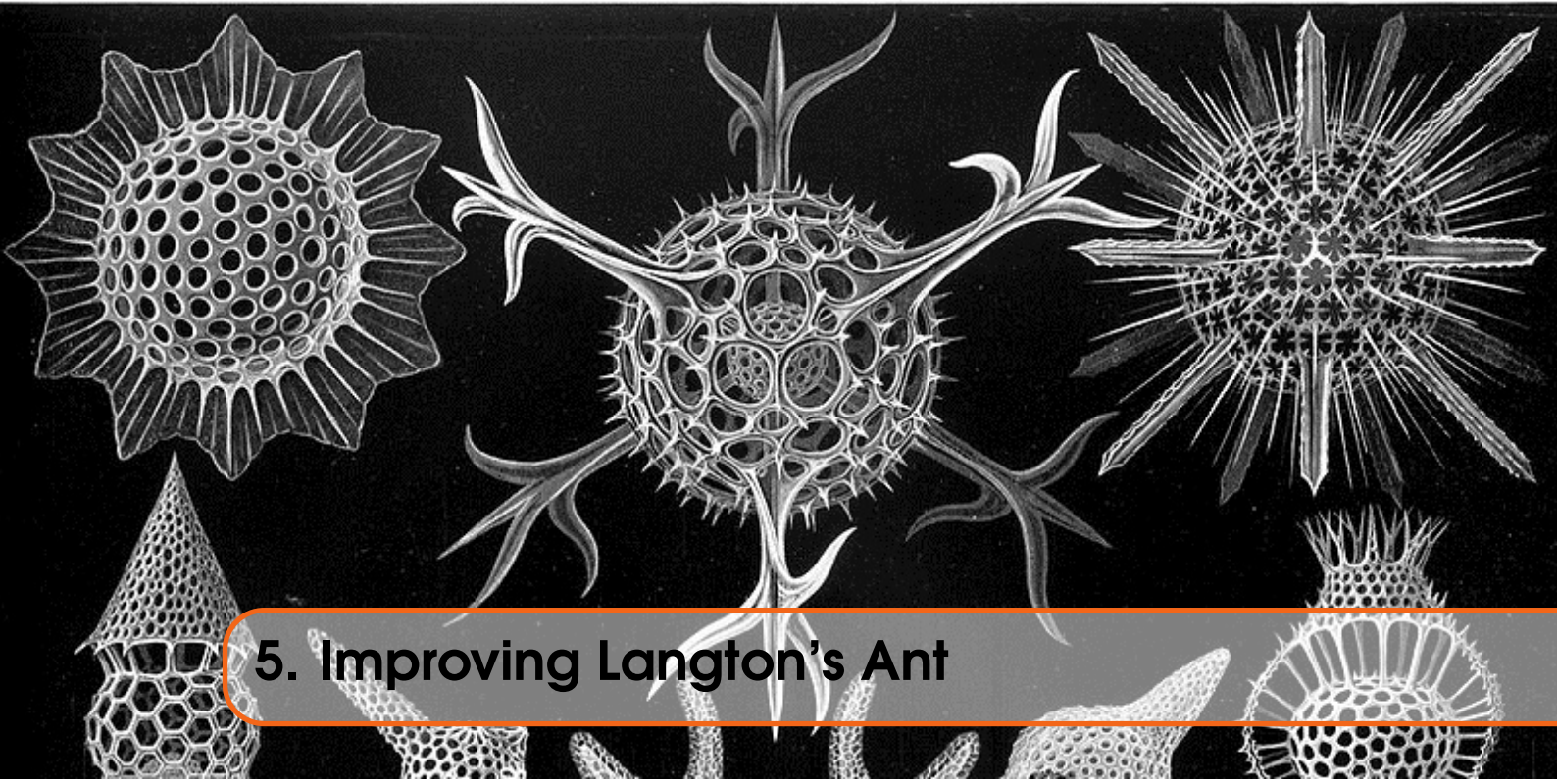
Given that this instruction has been removed from the MC68040 onwards, there's probably not much use for it in QL programs. See also `CALLM`.

The flags are set according to the word on the stack.

4.5.6 Coprocessor Instructions

There are numerous floating point co-processor instructions which, for now at least, are beyond the scope of the eComic![5] I'm pretty certain that George Gwilt, my faithful reader, has written a document on these instructions. The document can be found on the Sinclair QL Homepage.[6]

[5] I don't have a genuine MC680020 to play with, only QPC.
[6] `http://www.dilwyn.me.uk/docs/asm/fpu.zip`



5. Improving Langton's Ant

So, given that we now fully understand these new fangled 680020 instructions, can we improve Langton's Ant?

5.1 A 68020 Improvement, Perhaps?

As presented earlier in this issue, the code will actually assemble using the GWASL assembler that we have been using up until now in this series[1].

Given how short the code above is, relative to some of the stuff in the last issue for example, can it be improved using the 68020 processor?

In a word, yes. We have to work out a number of things each time through the loop:

- The address of the start of the row of bytes where the current cell can be found;
- The Address of the byte, within the row, representing the current cell;
- The Address of the bit, within the byte, representing the current cell;

We need this to enable us to determine the current cell's colour and from that where the next cell will be, plus we also need to change the colour to black or white according to whichever rule we activated.

The following code extract is where we work out the bit we need to set or clear in the ant's bitmap.

```
255 ;-------------------------------------------------------------------
256 ; First work out the bit number in the cell's byte. This is simply
257 ; the 7 - (across coordinate MOD 8).
258 ;-------------------------------------------------------------------
259 bitNumber
260             move.l   d2,d6                  ; Copy Across | Down coords
```

[1] And indeed, since I first started writing for *QL Today*, more years ago than I care to even begin to attempt to remember!

```
261             swap    d6                      ; Down | Across
262             move.w  d6,d4                   ; Copy Across coordinate to D4
263             andi.w  #7,d4                   ; Mod 8 = 0 to 7
264             neg.w   d4                      ; -7 to 0
265             addq.w  #7,d4                   ; D4.W = bit number
266
267 ;-------------------------------------------------------------------------
268 ; The byte within the row, which we calculate soon, is calculated as
269 ; (across coordinate DIV 8). Easy.
270 ;-------------------------------------------------------------------------
271 byteNumber
272             lsr.w   #3,d6                   ; Across div 8
273
274 ;-------------------------------------------------------------------------
275 ; The correct row number in the cells bitmap is the Down coordinate
276 ; multipled by (size_x DIV 8). This shouldn't exceed a word size. The
277 ; result is added to A3 which is the start address of the cells
278 ; bitmap and loaded into A4 as the byte address that we want.
279 ;-------------------------------------------------------------------------
280 doRow
281             move.w  #size_x,d7              ; Pixels across
282             lsr.w   #3,d7                   ; Now bytes across
283             mulu    d2,d7                   ; Times down coordinate
284             lea     0(a3,d7.w),a4           ; Address of correct row
285             adda.w  d6,a4                   ; Correct byte in row
286
287 ;-------------------------------------------------------------------------
288 ; Given the byte address in A4 and the bit number in D4, we can now
289 ; test that individual bit. A zero is white while a 1 is black.
290 ;-------------------------------------------------------------------------
291             btst    d4,(a4)                 ; 0 = white, 1 = black
```

Listing 5.1: Langtons Ant - Existing Bitmap Calculation

We can, should we wish to, replace all of the above using a bit field test instruction, or BFTST as described in BFTST - Test Bit Field on page 33.

Once you have read the introduction to bit fields - Bit Fields on page 28 you will realise that we no longer need to effectively reverse the bit numbering, as bit fields number from 0 upwards, but bit zero is the most significant bit!

Equally, we only need to calculate the bit number within the *entire* bitfield, we don't need to work out a bit number, a byte number and a row number first.

5.1.1 Bit Field Calculations

To calculate the bit number we need, is now quite simple - given that bit fields start numbering at the most significant bit. The calculation is:

$\text{across} + (\text{down} * \text{x\_size})$

That's it. If, for example we are on the very first bit, that would be bit zero in a bit field, the coordinates would both be 0, 0 for across and down, so this is $0 + (0 * 256)$ which is still zero, and is correct.

If we were at the far right of the very first line, we are on the 512^{th} bit in the bitmap, and that comes from the coordinates being 511, 0, giving $511 + (0 * 512)$ which is 511, and that is indeed the bit

number of the 512^{th} bit.

And so on down and across the bitmap. In our ant's world, there are 512 by 256 cells, so we need that many bits, which works out at 131,072 bits. (16,384 bytes)

This means that for the very bottom right coordinate, 511, 255, the bit will be $511 + (255 * 512)$ which works out at 131,071 and yes, that's the bit number of the $131,072^{nd}$ bit in the bitmap.

Currently we calculate a bit number in `D4` of the byte at the address held in `A4` for the current cell. Using the 68020 we can change all of the above code to the following.

```
255 ;------------------------------------------------------------------
256 ; Calculate the bit number within the cells bit field. This is simply
257 ; across + (down * width).
258 ;------------------------------------------------------------------
259 bitNumber
260             moveq   #x_size,d4          ; Width
261             mulu    d2,d4               ; Width * Down
262             swap    d2                  ; Low word = Across
263             add.w   d2,d4               ; Across + (Down * Width)
264             swap    d2                  ; Restore
265             bftst   (a3){d4:1}          ; Test bit
```

Listing 5.2: Langtons Ant - 68020 Bitmap Calculation

As you can see, this chops a whole pile of code out and reduces the number of instructions required to calculate which bit we need to be looking at and setting, or clearing, as part of the rules for the ant. It also frees up `D6` and `A4` too, which might come in handy elsewhere - for saving `D5` perhaps!

Obviously I now need to update the Register Usage comments - but I'm not showing that here.

Register `A3`, if you remember, was initially set to point at the `Cells` bitmap. So the `bftst (a3){d4:1}` instruction says to *start at the address held in register `A3` and count from the most significant bit at that address, `D4` bits along, then test the single bit that you arrive at.*

After this, we also need to update the code for the two rules as we must set or clear the bit representing the current cell. The existing code for the two rules is as follows.

```
293 ;------------------------------------------------------------------
294 ; Rule 1: If the cell is white, turn it black, right turn, walk on.
295 ;------------------------------------------------------------------
296 ruleOne
297             bne.s   ruleTwo
298             bset    d4,(a4)             ; Turn it black in the bitmap
299             moveq   #black,d1           ; Colour for BLOCK command
300             subq.b  #1,d5               ; Direction - 1 = right
301             bra.s   doDirection         ; Prepare to walk on
```

Listing 5.3: Langtons Ant - Existing Rule 1

```
303 ;------------------------------------------------------------------
304 ; Rule 2: If the cell is black, turn it white, left turn, walk on.
305 ;------------------------------------------------------------------
306 ruleTwo
307             bclr    d4,(a4)             ; Turn it white in the bitmap
308             moveq   #white,d1           ; Colour for BLOCK command
309             addq.b  #1,d5               ; Direction + 1 = left
```

Listing 5.4: Langtons Ant - Existing Rule 2

These would be changed to allow them to use the BFSET and BFCLR instructions, as follows:

```
293 ;-------------------------------------------------------------------------
294 ; Rule 1: If the cell is white, turn it black, right turn, walk on.
295 ;-------------------------------------------------------------------------
296 ruleOne
297             bne.s   ruleTwo
298             bfset   (a3){d4:1}          ; Turn it black in the bitmap
299             moveq   #black,d1           ; Colour for BLOCK command
300             subq.b  #1,d5               ; Direction - 1 = right
301             bra.s   doDirection         ; Prepare to walk on
```

Listing 5.5: Langtons Ant - 68020 Rule 1

```
303 ;-------------------------------------------------------------------------
304 ; Rule 2: If the cell is black, turn it white, left turn, walk on.
305 ;-------------------------------------------------------------------------
306 ruleTwo
307             bfclr   (a3){d4:1}          ; Turn it white in the bitmap
308             moveq   #white,d1           ; Colour for BLOCK command
309             addq.b  #1,d5               ; Direction + 1 = left
```

Listing 5.6: Langtons Ant - 68020 Rule 2

So a couple of easy changes and we have a smaller source program to type in, a smaller executable program - in my case, by a whole 34 bytes - because we took advantage of the 68020's new instructions.

Happy anting.

6. This eMagazine is now on Github

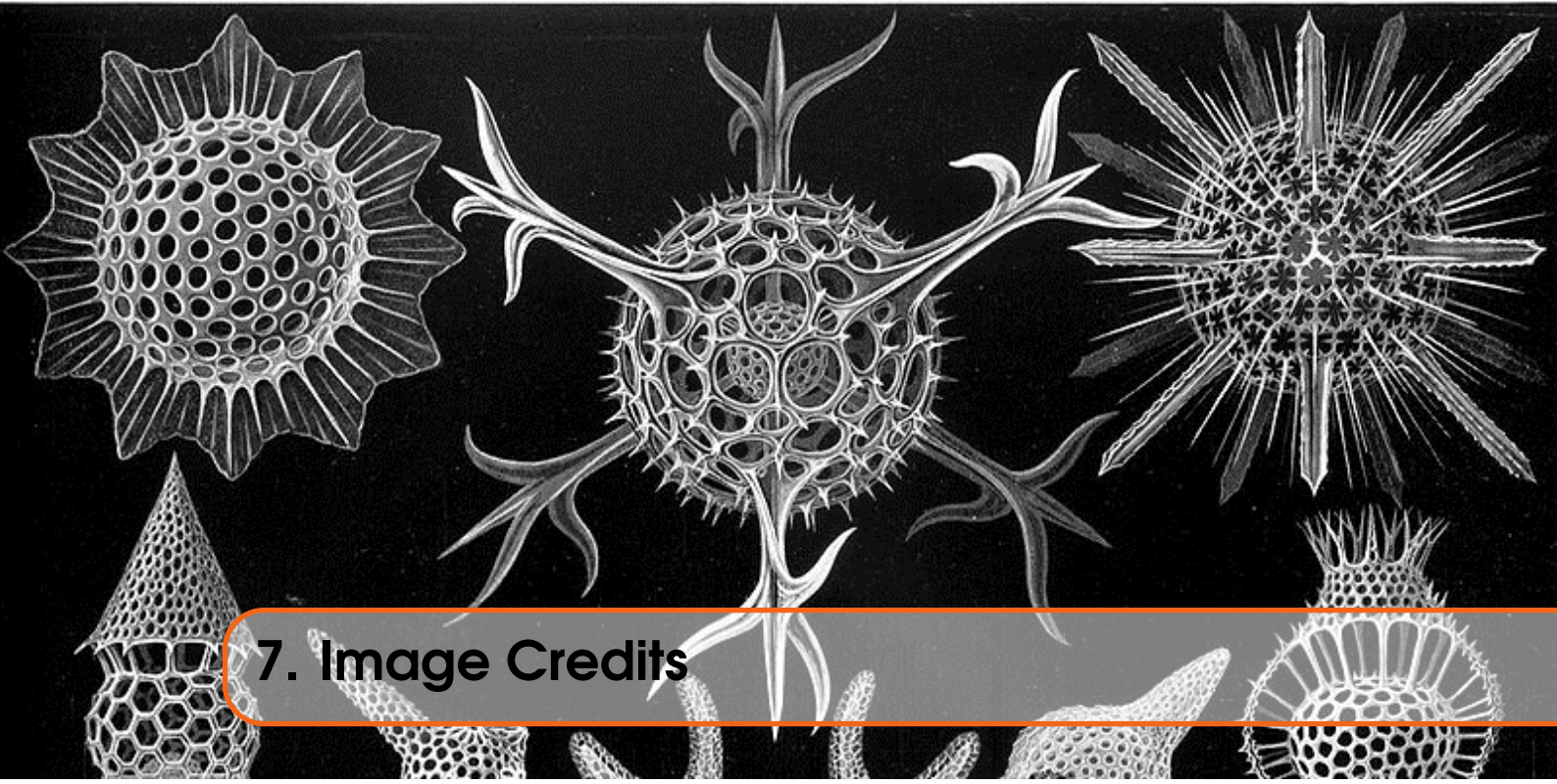
As of a few days/weeks/months or even years ago, depending on when you read this, the source code for the magazine[1] can be found on https://github.com.

If you go there (by clicking the link above) then you will find all the previous issues, except for the first one. That one was created using a different system very unlike the current system, and I have not yet converted it to use latex. Maybe one day I will - time permitting.

Hopefully, I'll get the source code for the various listings explained in the eMagazine, up there soon too.

Have fun.

[1] But not *yet* the sources for the listings.



7. Image Credits

The front cover image on this ePeriodical is taken from the book *Kunstformen der Natur* by German biologist Ernst Haeckel. The book was published between 1899 and 1904. The image used is of various *Polycystines* which are a specific kind of micro-fossil.

I have also cropped the image for use on each chapter heading page.

You can read about Polycystines on Wikipedia and there is a brief overview of the above book, also on Wikipedia, which shows a number of other images taken from the book. (Some of which I considered before choosing the current one!)

Polycystines have absolutely nothing to do with the QL or computing in general - in fact, I suspect they died out before electricity was invented - but I liked the image, and decided that it would make a good cover for the book and a decent enough chapter heading image too.

Not that I am suggesting, *in any way whatsoever*, that we QL fans are ancient.